Programming Manual

Allen-Bradley

# Logix5000 Controllers Add-On Instructions

Catalog Numbers 1756 ControlLogix, 1756 GuardLogix, 1768 CompactLogix, 1768 Compact GuardLogix, 1769 CompactLogix, 1789 SoftLogix, PowerFlex with DriveLogix

Allen-Bradley • Rockwell Software

Rockwell Automation

## Important User Information

Solid-state equipment has operational characteristics differing from those of electromechanical equipment. Safety Guidelines for the Application, Installation and Maintenance of Solid State Controls (publication SGI-1.1 available from your local Rockwell Automation sales office or online at http://www.rockwellautomation.com/literature/) describes some important differences between solid-state equipment and hard-wired electromechanical devices. Because of this difference, and also because of the wide variety of uses for solid-state equipment, all persons responsible for applying this equipment must satisfy themselves that each intended application of this equipment is acceptable.

In no event will Rockwell Automation, Inc. be responsible or liable for indirect or consequential damages resulting from the use or application of this equipment.

The examples and diagrams in this manual are included solely for illustrative purposes. Because of the many variables and requirements associated with any particular installation, Rockwell Automation, Inc. cannot assume responsibility or liability for actual use based on the examples and diagrams.

No patent liability is assumed by Rockwell Automation, Inc. with respect to use of information, circuits, equipment, or software described in this manual.

Reproduction of the contents of this manual, in whole or in part, without written permission of Rockwell Automation, Inc., is prohibited.

Throughout this manual, when necessary, we use notes to make you aware of safety considerations.

| | |
|---|---|
| | **WARNING:** Identifies information about practices or circumstances that can cause an explosion in a hazardous environment, which may lead to personal injury or death, property damage, or economic loss. |
| | **ATTENTION:** Identifies information about practices or circumstances that can lead to personal injury or death, property damage, or economic loss. Attentions help you identify a hazard, avoid a hazard, and recognize the consequence |
| | **SHOCK HAZARD:** Labels may be on or inside the equipment, for example, a drive or motor, to alert people that dangerous voltage may be present. |
| | **BURN HAZARD:** Labels may be on or inside the equipment, for example, a drive or motor, to alert people that surfaces may reach dangerous temperatures. |
| **IMPORTANT** | Identifies information that is critical for successful application and understanding of the product. |

Allen-Bradley, Rockwell Software, Rockwell Automation, and TechConnect are trademarks of Rockwell Automation, Inc.

Trademarks not belonging to Rockwell Automation are property of their respective companies.

This manual contains new and updated information.

**IMPORTANT** RSLogix 5000 programming software is now known as Studio 5000™ Logix Designer application, a component of Studio 5000 Engineering and Design Environment.

The following controllers are no longer supported in the Logix Designer application, version 21.

| Catalog Number | Description |
|---|---|
| 1756-L61 | ControlLogix 5561 Controller |
| 1756-L61S | ControlLogix 5561S Controller |
| 1756-L62 | ControlLogix 5562 Controller |
| 1756-L62S | ControlLogix 5562S Controller |
| 1756-L63 | ControlLogix 5563 Controller |
| 1756-L63S | ControlLogix 5563S Controller |
| 1756-L64 | ControlLogix 5564 Controller |
| 1756-L65 | ControlLogix 5565 Controller |
| 1768-L43 | CompactLogix 5343 Controller |
| 1768-L43S | CompactLogix 5343S Controller |
| 1768-L45 | CompactLogix 5345 Controller |
| 1768-L45S | CompactLogix 5345S Controller |
| 1769-L23E-QBF1 | CompactLogix 5323E-QB1 Controller |
| 1769-L23E-QBFC1 | CompactLogix 5323E-QBFC1 Controller |
| 1769-L23-QBFC1 | CompactLogix 5323-QBFC1 Controller |
| 1769-L31 | CompactLogix 5331 Controller |
| 1769-L32C | CompactLogix 5332C Controller |
| 1769-L32E | CompactLogix 5332E Controller |
| 1769-L35CR | CompactLogix 5335CR Controller |
| 1769-L35E | CompactLogix 5335E Controller |

Changes throughout this revision are marked by change bars, as shown in the margin of this page.

This table contains the changes made to this revision.

| Topic | Page |
|---|---|
| About Add-On Instructions | 11 |
| Data Type | 15 |
| Help | 18 |
| Instruction Size | 22 |
| REAL data types are allowed for 1756-L7xS controllers. | 37 |
| Project Documentation | 63 |

## Notes:

**Preface**

Studio 5000 Engineering and Design Environment and Logix Designer Application . . . . . 7
Purpose of This Manual . . . . . 7
Additional Resources for Safety Applications . . . . . 7
Understanding Terminology . . . . . 9

**Chapter 1**

**Designing Add-On Instructions**

Introduction . . . . . 11
About Add-On Instructions . . . . . 11
Components of an Add-On Instruction . . . . . 13
General Information . . . . . 13
Parameters . . . . . 14
Local Tags . . . . . 14
Data Type . . . . . 15
Logic Routine . . . . . 15
Optional Scan Mode Routines . . . . . 16
Instruction Signature . . . . . 16
Signature History . . . . . 17
Change History . . . . . 17
Help . . . . . 18
Considerations for Add-On Instructions . . . . . 18
Instruction Functionality . . . . . 18
Encapsulation . . . . . 19
Safety Add-On Instructions . . . . . 19
Instruction Signature . . . . . 20
Safety Instruction Signature . . . . . 21
Programming Languages . . . . . 21
Transitional Instructions . . . . . 21
Instruction Size . . . . . 22
Runtime Editing . . . . . 22
Nesting Add-On Instructions . . . . . 22
Routines Versus Add-On Instructions . . . . . 23
Programmatic Access to Data . . . . . 24
Unavailable Instructions within Add-On Instructions . . . . . 24
Using GSV and SSV Instructions . . . . . 26
Considerations When Creating Parameters . . . . . 27
Passing Arguments to Parameters by Reference or by Value . . . . . 27
Selecting a Data Type for a Parameter . . . . . 27
Creating an Alias Parameter for a Local Tag . . . . . 28
Using a Single Dimension Array as an InOut Parameter . . . . . 28
Determining Which Parameters to Make Visible or Required . . . . 29
Using Standard and Safety Tags . . . . . 30
Data Access Control . . . . . 30
Planning Your Add-On Instruction Design . . . . . 32
Intended Behavior . . . . . 32

Parameters . . . . . 32
Naming Conventions . . . . . 32
Source Protection . . . . . 32
Nesting - Reuse Instructions . . . . . 33
Local Tags . . . . . 33
Programming Languages . . . . . 33
Scan Mode Routines . . . . . 34
Test . . . . . 34
Help Documentation . . . . . 34

## Chapter 2

## Defining Add-On Instructions

Introduction . . . . . 35
Creating an Add-On Instruction . . . . . 35
Creating a Parameter . . . . . 37
Creating Local Tags . . . . . 39
Editing Parameters and Local Tags . . . . . 41
Updates to Arguments Following Parameter Edits . . . . . 41
Copying Parameter or Local Tag Default Values . . . . . 43
Creating Logic for the Add-On Instruction . . . . . 44
Execution Considerations for Add-On Instructions . . . . . 44
Optimizing Performance . . . . . 45
Defining Operation in Different Scan Modes . . . . . 45
Enabling Scan Modes . . . . . 46
Prescan Routine . . . . . 46
Postscan Routine . . . . . 48
EnableInFalse Routine . . . . . 50
Using the EnableIn and EnableOut Parameters . . . . . 51
EnableIn Parameter and Ladder Diagram . . . . . 52
EnableIn Parameter and Function Blocks . . . . . 52
EnableIn Parameter and Structured Text . . . . . 52
Changing the Class of an Add-On Instruction . . . . . 53
Testing the Add-On Instruction . . . . . 53
Before You Test . . . . . 54
Testing the Flow . . . . . 54
Monitoring Logic with Data Context Views . . . . . 54
Verifying Individual Scan Modes . . . . . 55
Defining Source Protection for an Add-On Instruction . . . . . 55
Enable the Source Protection Feature . . . . . 56
Generating an Add-On Instruction Signature . . . . . 56
Generating, Removing, or Copying an Instruction Signature . . . . . 56
Creating a Signature History Entry . . . . . 57
Generating a Safety Instruction Signature . . . . . 58
Viewing and Printing the Instruction Signature . . . . . 58
Creating Instruction Help . . . . . 59
Writing Clear Descriptions . . . . . 59

Documenting an Add-On Instruction . . . . . 61
Project Documentation . . . . . 63
Motor Starter Instruction Example . . . . . 64
Simulation Instruction Example . . . . . 66
Ladder Diagram Configuration . . . . . 68
Function Block Diagram Configuration . . . . . 69
Structured Text Configuration . . . . . 69

**Chapter 3**

**Using Add-On Instructions**

Introduction . . . . . 71
Accessing Add-On Instructions . . . . . 71
Using the Add Element Dialog Box . . . . . 72
Including an Add-On Instruction in a Routine . . . . . 73
Tips for Using an Add-On Instruction . . . . . 75
Programmatically Access a Parameter . . . . . 75
Using the Jog Command in Ladder Diagram . . . . . 76
Using the Jog Command In Function Block Diagram . . . . . 76
Using the Jog Command in Structured Text . . . . . 77
Monitoring the Value of a Parameter . . . . . 77
Viewing Logic and Monitor with Data Context . . . . . 79
Determining if the Add-On Instruction is Source Protected . . . . . 81
Copying an Add-On Instruction . . . . . 82
Storing Your Add-On Instructions . . . . . 82

**Chapter 4**

**Importing and Exporting Add-On Instructions**

Introduction . . . . . 85
Creating an Export File . . . . . 85
Exporting to Separate Files . . . . . 86
Exporting to a Single File . . . . . 87
Importing an Add-On Instruction . . . . . 88
Import Considerations . . . . . 88
Import Configuration . . . . . 89
Updating an Add-On Instruction to a Newer Revision via Import . . . . 90

**Index**

## Studio 5000 Engineering and Design Environment and Logix Designer Application

The Studio 5000™ Engineering and Design Environment combines engineering and design elements into a common environment. The first element in the Studio 5000 environment is the Logix Designer application. The Logix Designer application is the rebranding of RSLogix™ 5000 software and will continue to be the product to program Logix5000™ controllers for discrete, process, batch, motion, safety, and drive-based solutions.

The Studio 5000 environment is the foundation for the future of Rockwell Automation® engineering design tools and capabilities. It is the one place for design engineers to develop all the elements of their control system.

## Purpose of This Manual

This manual shows how to design, configure, and program Add-On Instructions. This manual is one of a set of related manuals that show common procedures for programming and operating Logix5000 controllers. For a complete list of common procedures manuals, see the *Logix5000 Controllers Common Procedures Programming Manual*, publication 1756-PM001.

The term Logix5000 controller refers to any controller that is based on the Logix5000 operating system, including the following:

- CompactLogix and Compact GuardLogix controllers
- ControlLogix and GuardLogix controllers
- DriveLogix controllers
- FlexLogix controllers
- SoftLogix5800 controllers

## Additional Resources for Safety Applications

With the addition of safety Add-On Instructions in RSLogix 5000 software, version 18, you can use Add-On Instructions in GuardLogix and Compact GuardLogix safety-related functions. This manual provides information about safety Add-On Instructions.

For detailed information on safety application requirements, the safety task signature, and configuring and operating safety controllers, refer to these publications:

- *GuardLogix Controller Systems Safety Reference Manual*, publication 1756-RM093
- *Compact GuardLogix Controllers User Manual*, publication 1768-UM002
- *GuardLogix Controllers User Manual*, publication 1756-UM020
- *GuardLogix Safety Application Instruction Set Safety Reference Manual*, publication 1756-RM095

## Understanding Terminology

This table defines some of the terms used in this manual when describing how parameters and arguments are used in Add-On Instructions.

| Term | Definition |
|---|---|
| Argument | An argument is assigned to a parameter of an Add-On Instruction instance. An argument contains the specification of the data used by an instruction in a user program. An argument can contain the following:<br>•A simple tag (for example, L101)<br>•A literal value (for example, 5)<br>•A tag structure reference (for example, Recipe.Temperature)<br>•A direct array reference (for example, Buffer[1])<br>•An indirect array reference (for example, Buffer[Index+1])<br>•A combination (for example, Buffer[Index+1].Delay) |
| Parameter | Parameters are created in the Add-On Instruction definition. When an Add-On Instruction is used in application code, arguments must be assigned to each required parameter of the Add-On Instruction. |
| InOut parameter | An InOut parameter defines data that can be used as both input and output data during the execution of the instruction. Because InOut parameters are always passed by reference, their values can change from external sources during the execution of the Add-On Instruction. |
| Input parameter | For an Add-On Instruction, an Input parameter defines the data that is passed by value into the executing instruction. Because Input parameters are always passed by value, their values cannot change from external sources during the execution of the Add-On Instruction. |
| Output parameter | For an Add-On Instruction, an Output parameter defines the data that is produced as a direct result of executing the instruction. Because Output parameters are always passed by value, their values cannot change from external sources during the execution of the Add-On Instruction. |
| Passed by reference | When an argument is passed to a parameter by reference, the logic directly reads or writes the value that the tag uses in controller memory. Because the Add-On Instruction is acting on the same tag memory as the argument, other code or HMI interaction that changes the argument's value can change the value while the Add-On Instruction is executing. |
| Passed by value | When an argument is passed to a parameter by value, the value is copied in or out of the parameter when the Add-On Instruction executes. The value of the argument does not change from external code or HMI interaction outside of the Add-On Instruction itself. |

# Designing Add-On Instructions

## Introduction

Add-On Instructions are available beginning with RSLogix 5000 software, version 16. Add-On Instructions are custom instructions that you design and create. Beginning in RSLogix 5000 software, version 18, high integrity and safety Add-On Instructions are available.


| Topic | Page |
|---|---|
| About Add-On Instructions | 11 |
| Components of an Add-On Instruction | 13 |
| Considerations for Add-On Instructions | 18 |
| Considerations When Creating Parameters | 27 |
| Planning Your Add-On Instruction Design | 32 |


## About Add-On Instructions

With Add-On Instructions, you can create new instructions for sets of commonly-used logic, provide a common interface to this logic, and provide documentation for the instruction.

Add-On Instructions are intended to be used to encapsulate commonly used functions or device control. They are not intended to be a high-level hierarchical design tool. Programs with routines are better suited to contain code for the area or unit levels of your application.

These are some benefits to using Add-On Instructions:

- Reuse code
  - You can use Add-On Instructions to promote consistency between projects by reusing commonly-used control algorithms.
  - If you have an algorithm that will be used multiple times in the same project or across multiple projects, it may make sense to incorporate that code inside an Add-On Instruction to make it modular and easier to reuse.
- Provide an easier to understand interface
  - You can place complicated algorithms inside of an Add-On Instruction, and then provide an easier to understand interface by making only essential parameters visible or required.
  - You can reduce documentation development time through automatically generating instruction help.
- You can export an Add-On Instruction to an .L5X file that can then be imported into another project. You can also copy and paste between projects.

- Protect intellectual property
  - You can place your proprietary code inside of an Add-On Instruction, then use Source Protection to prevent others from viewing or changing your code.
- Simplify maintenance
  - You can simplify code maintenance because Add-On Instruction logic, monitored in the Logix Designer application, animates with tag values relative to that specific instance of the Add-On Instruction.
- Track revisions, a chang history, and easily confirm instruction functionality with high-integrity Add-On Instructions
  - You can add an instruction signature to your Add-On Instruction, which generates a unique identifier and prevents the instruction from being edited without resulting in a change to the signature.

An Add-On-Instruction can be used across multiple projects. You can define the instructions, the instructions can be provided to you by someone else, or they can be copied from another project.

Once defined in a project, they behave similarly to the built-in instructions already available in the Logix Designer application. They appear on the instruction toolbar and in the instruction browser for easy access, just like built-in Logix Designer application instructions.

Like standard Add-On Instructions, safety Add-On Instructions let you encapsulate commonly-used safety logic into a single instruction, making it modular and easier to reuse. In addition to the instruction signature used for high-integrity Add-On Instructions, safety Add-On Instructions feature a SIL 3 safety instruction signature for use in safety-related functions up to and including SIL 3.

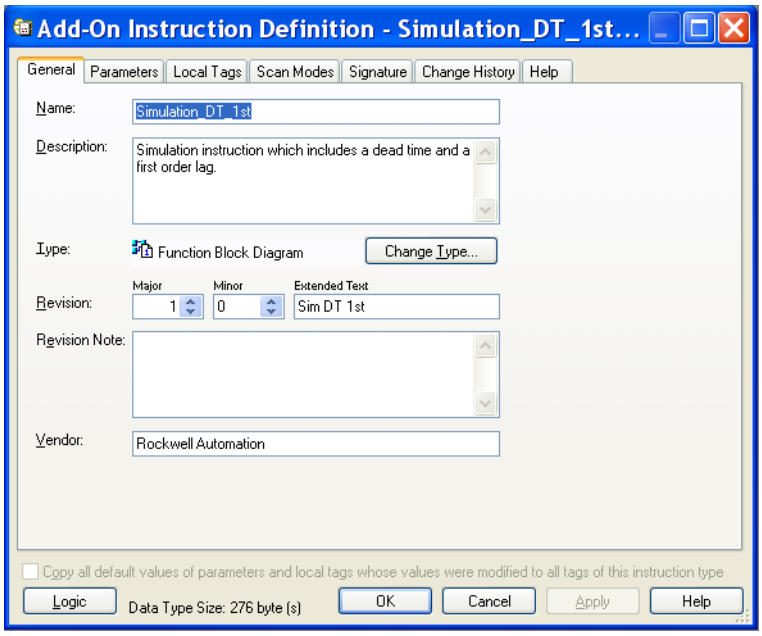
Refer to the *GuardLogix Controller Systems Safety Reference Manual*, publication 1756-RM093, for details on certifying safety Add-On Instructions and using them in SIL 3 safety applications.

## Components of an Add-On Instruction

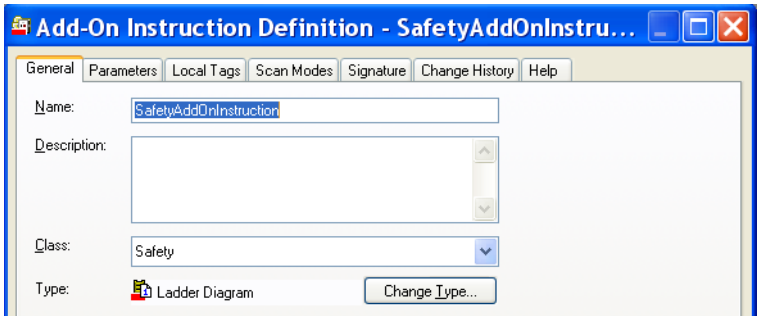
Add-On Instructions are made up of the following parts.

### General Information

The General tab contains the information you enter when you first create the instruction. You can use this tab to update that information. The description, revision, revision note, and vendor information is copied into the custom help for the instruction. The revision is not automatically managed by the software. You are responsible for defining how it is used and when it is updated.

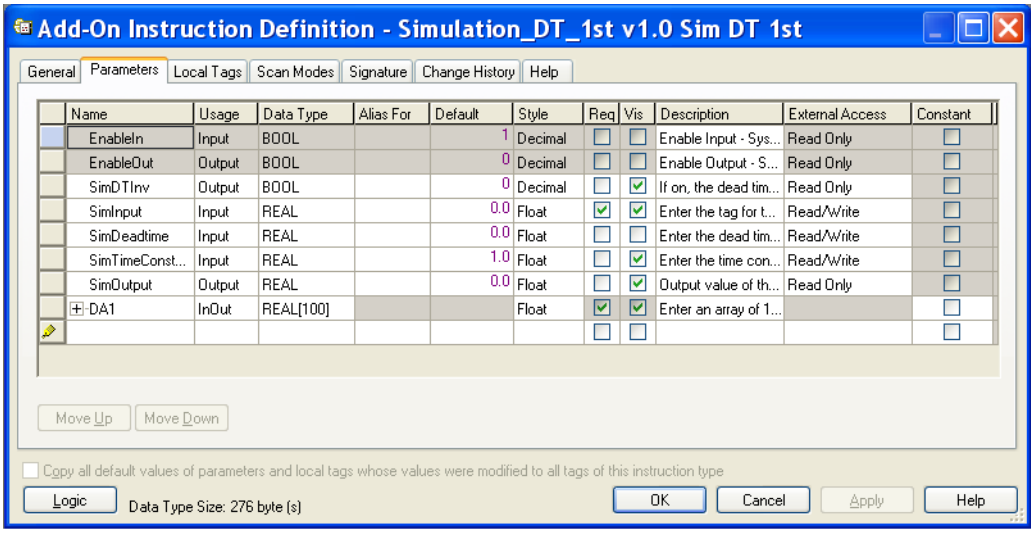



Class information for safety controller projects appears on the General tab as well. The class can be standard or safety. Safety Add-On Instructions must meet requirements specific to safety applications. See "Safety Add-On Instructions" on page 19 for more information.



## Parameters

The parameters define the instruction interface; that is, how the instruction appears when used. The parameter order defines the order that the parameters appear on the instruction call.

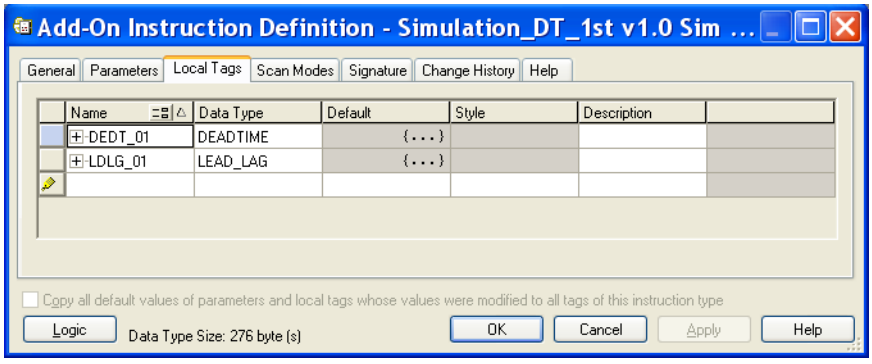



## Local Tags

The Local Tags tab defines tags that are used by the logic within the Add-On Instruction and are not visible outside the instruction. Other Add-On Instructions or programs in the project cannot access these tags.

The only way to make a local tag or one of its members accessible from outside the instruction is by defining an alias parameter.

See "Creating an Alias Parameter for a Local Tag" on page 28.



## Data Type

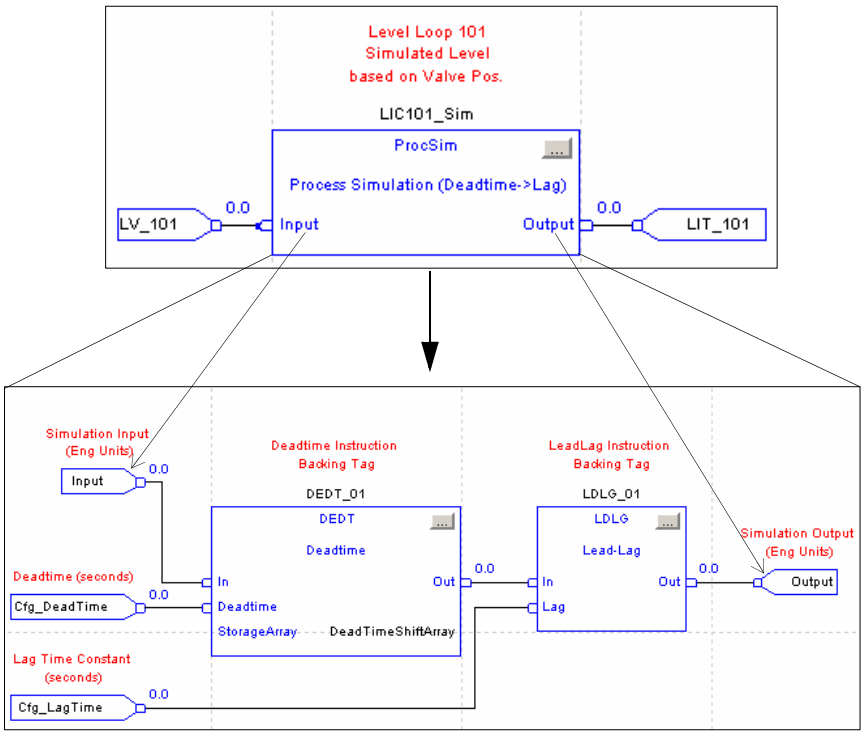
Parameters and local tags are used to define the data type that is used when executing the instruction. The software builds the associated data type. The software orders the members of the data type that correspond to the parameters in the order that the parameters are defined. Local tags are added as hidden members.

The data type for a Local Tag may not be a multi-dimensional array or an object type, which includes all Motion types, MSG, ALARM_ANALOG, and ALARM_DIGITAL.

The data type field is unavailable for members of a Local Tag.

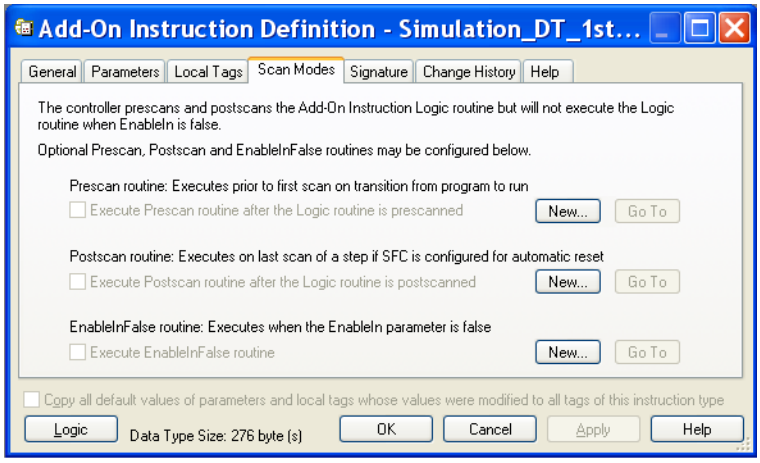
## Logic Routine

The logic routine of the Add-On Instruction defines the primary functionality of the instruction. It is the code that executes whenever the instruction is called. Shown below is the interface of an Add-On Instruction and its primary logic routine that defines what the instruction does.



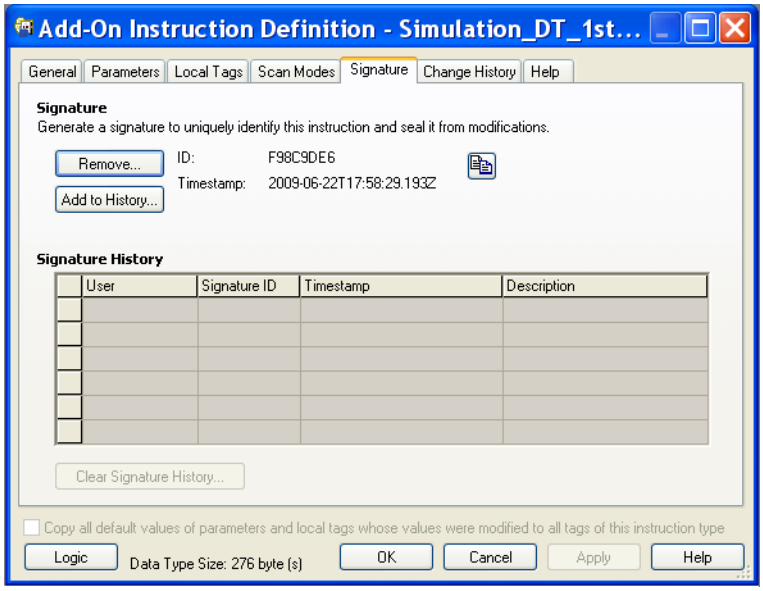
## Optional Scan Mode Routines

You can define additional routines for Scan mode behavior.


Add-On Instruction Definition - Simulation_DT_1st...
General
Parameters
Local Tags
Scan Modes
Signature
Change History
Help
The controller prescans and postscans the Add-On Instruction Logic routine but will not execute the Logic routine when EnableIn is false.
Optional Prescan, Postscan and EnableInFalse routines may be configured below.
Prescan routine: Executes prior to first scan on transition from program to run
Execute Prescan routine after the Logic routine is prescanned
New...
Go To
Postscan routine: Executes on last scan of a step if SFC is configured for automatic reset
Execute Postscan routine after the Logic routine is postscanned
New...
Go To
EnableInFalse routine: Executes when the EnableIn parameter is false
Execute EnableInFalse routine
New...
Go To
Copy all default values of parameters and local tags whose values were modified to all tags of this instruction type
Logic
Data Type Size: 276 byte (s)
OK
Cancel
Apply
Help

## Instruction Signature

The instruction signature consists of an ID number that identifies the contents of the Add-On Instruction and a timestamp that identifies the specific date and time at which the instruction signature was generated or a signature history entry was made (whichever came last).




Once generated, the instruction signature seals the Add-On Instruction, preventing it from being edited while the signature is in place.

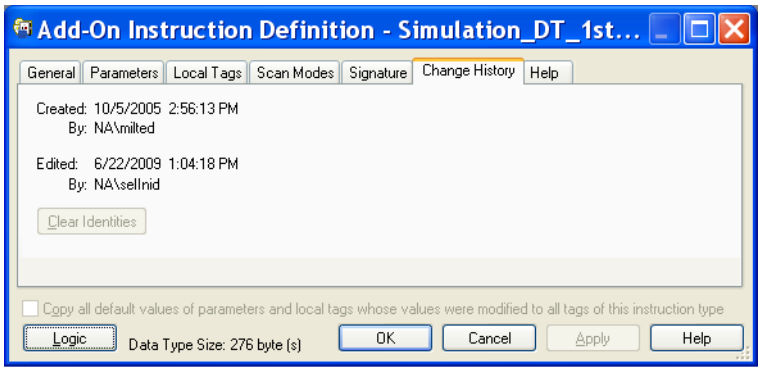
In addition, when a sealed safety Add-On Instruction is downloaded for the first time, a SIL 3 safety instruction signature is automatically generated. The safety instruction signature is an ID number that identifies the execution characteristics of the safety Add-On Instruction.

## Signature History

The signature history provides a record of signatures for future reference. A signature history entry consists of the name of the user, the instruction signature, the timestamp value, and a user-defined description. Up to six history entries can be stored. If a seventh entry is made, the oldest entry is automatically deleted.

## Change History

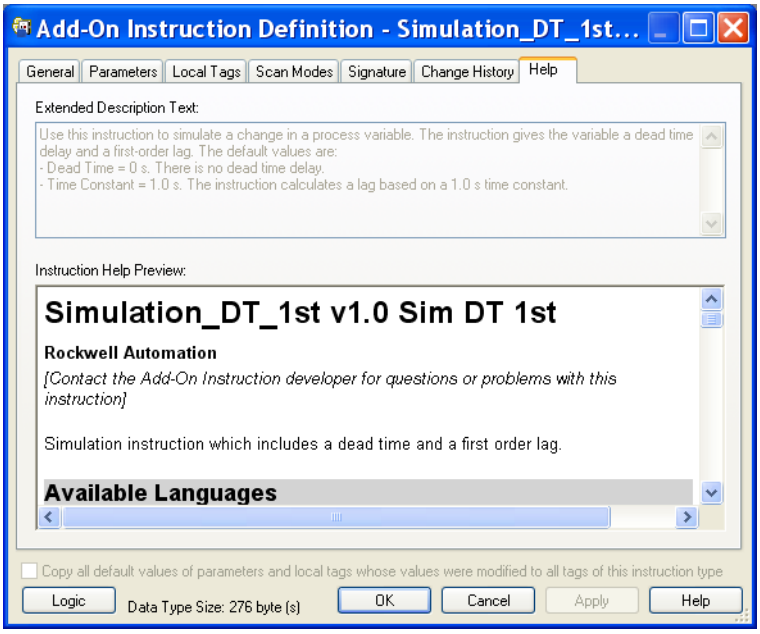
The Change History tab displays the creation and latest edit information that is tracked by the software. The By fields show who made the change based on the Windows user name at the time of the change.



### Help

The name, revision, description, and parameter definitions are used to automatically build the Instruction Help. Use the Extended Description Text to provide additional Help documentation for the Add-On Instruction. This content reflects updates as Parameters or other attributes are changed in the definition.

The Instruction Help Preview shows how your instruction will appear in the various languages, based on parameters defined as required or visible.


Add-On Instruction Definition - Simulation_DT_1st...
General
Parameters
Local Tags
Scan Modes
Signature
Change History
Help
Extended Description Text:
Use this instruction to simulate a change in a process variable. The instruction gives the variable a dead time delay and a first-order lag. The default values are:
· Dead Time = 0 s. There is no dead time delay.
· Time Constant = 1.0 s. The instruction calculates a lag based on a 1.0 s time constant.
Instruction Help Preview:
Simulation_DT_1st v1.0 Sim DT 1st
Rockwell Automation
[Contact the Add-On Instruction developer for questions or problems with this instruction]
Simulation instruction which includes a dead time and a first order lag.
Available Languages
Copy all default values of parameters and local tags whose values were modified to all tags of this instruction type
Logic
Data Type Size: 276 byte (s)
OK
Cancel
Apply
Help

## Considerations for Add-On Instructions

When deciding whether to develop an Add-On Instruction, consider the following aspects.

### Instruction Functionality

Complex instructions tend to be highly application specific and not reusable, or require extensive configuration support code. As with the built-in instructions, Add-On Instructions need to do one thing, do it well, and support modular coding. Consider how the instruction will be used and manage interface complexity for the end user or application.

Add-On Instructions are best at providing a specific type of functionality or device control.

## Encapsulation

Add-On Instructions are designed to fully encapsulate the code and data associated with the instruction. The logic inside an Add-On Instruction uses only the parameters and local tags defined by the instruction definition. There is no direct programmatic access to controller or program scope tags. This lets the Add-On Instruction be a standalone component that can execute in any application that calls it by using the parameters interface. It can be validated once and then locked to prevent edits.

## Safety Add-On Instructions

Safety Add-On Instructions are used in the safety task of GuardLogix safety controllers. Create a safety Add-On Instruction if you need to use your instruction in a safety application. Safety Add-On Instructions are subject to a number of restrictions. These restrictions, enforced by the Logix Designer application and all GuardLogix controllers, are listed here for informational purposes only.

- They may use only safety-approved instructions and data types.
- All parameters and local tags used in a safety Add-On Instruction must also be safety class.
- Safety Add-On Instructions use relay ladder logic only and can be called in safety routines only, which are currently restricted to ladder logic.
- Safety Add-On Instructions may be referenced by other safety Add-On Instructions, but not by standard Add-On Instructions.
- Safety Add-On instructions cannot be created, edited, or imported when a safety project is safety-locked or has a safety task signature.

Refer to the *GuardLogix Controller Systems Safety Reference Manual*, publication 1756-RM093, for information on how to certify a safety Add-On Instruction as well as details on requirements for safety applications, the safety task signature, and a list of approved instructions and data types.

## Instruction Signature

The instruction signature, available for both standard and safety controllers, lets you quickly determine if the Add-On Instruction has been modified. Each Add-On Instruction has its own instruction signature on the Add-On Instruction definition. The instruction signature is required when an Add-On Instruction is used in SIL 3 safety-related functions, and may be required for regulated industries. Use it when your application calls for a higher level of integrity.

Once generated, the instruction signature seals the Add-On Instruction, preventing it from being edited until the signature is removed. This includes rung comments, tag descriptions, and any instruction documentation that was created. When an instruction is sealed, you can perform only these actions:

- Copy the instruction signature
- Create or copy a signature history entry
- Create instances of the Add-On Instruction
- Download the instruction
- Remove the instruction signature
- Print reports

The instruction signature does not prevent referenced Add-On Instructions or User-defined Data Types from being modified. Changes to the parameters of a referenced Add-On Instruction or to the members of a referenced User-defined Data Type can cause the instruction signature to become invalid. These changes include:

- Adding, deleting, or moving parameters, local tags, or members in referenced User-defined Data Types.
- Changing the name, data type, or display style of parameters, local tags, or members in referenced User-defined Data Types.

If you want to enable project documentation or source protection on an Add-On Instruction that will be sealed with an instruction signature, you need to import the translated information or apply source protection before generating the signature. You must have the source key to generate a signature or to create a signature history entry for a source-protected Add-On Instruction that has an instruction signature.

See "Defining Source Protection for an Add-On Instruction" on page 55 for more information on source protecting your Add-On Instruction.

## Safety Instruction Signature

When a sealed safety Add-On Instruction is downloaded for the first time, a SIL 3 safety instruction signature is automatically generated.

For details on how to certify a safety Add-On Instruction, refer to the *GuardLogix Controller Systems Safety Reference Manual*, publication 1756-RM093.

## Programming Languages

Select the programming language based on the type of application you are developing. Ladder Diagram, Function Block Diagram, and Structured Text can be used for Add-On Instruction logic.

Each of the programming languages supported in the Logix Designer application is targeted for different types of applications and programming styles. In general, Ladder Diagram executes simple boolean logic, timers, and counters the fastest. Function Block Diagrams and Structured Text may be more efficient if you take advantage of the more advanced process and drives instructions available in those languages.

You cannot compare execution times for the same Add-On Instruction written in different programming languages. There are fundamental differences on how the different languages execute and are compiled.

TIP You can change the programming language after you create the Add-On Instruction by clicking **Change Type** on the General tab of the Add-On Instruction Definition Editor. However, any existing logic will be lost.

## Transitional Instructions

Some instructions execute (or retrigger) only when rung-condition-in toggles from false to true. These are transitional-relay ladder instructions. When used in an Add-On Instruction, these instructions will not detect the rung-in transition to the false state. When the EnableIn bit is false, the Add-On Instruction logic routine no longer executes, thus the transitional instruction does not detect the transition to the false state. Extra conditional logic is required to handle triggering of transitional instructions contained in an Add-On Instruction.

Some examples of transitional instructions include: ONS, MSG, PXRQ, SRT, some of the ASCII instructions, and some of the Motion instructions.

TIP The EnableInFalse routine can be used to provide the conditioning required to retrigger transitional instructions contained in an Add-On Instruction. However, this method will not work for calls to this Add-On Instruction contained in a Structured Text routine, since EnableIn is always true for calls in Structured Text.

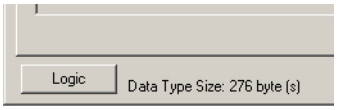
## Instruction Size

Add-On Instructions have one primary logic routine that defines the behavior of the instruction when executed. This logic routine is like any other routine in the project and has no additional restrictions in length. The total number of Input parameters plus Output parameters plus local tags can be up to 512.

There are no limits on the number of InOut parameters. However, limit extended properties (.@Min and .@Max syntax) should not be defined on an InOut parameter of an Add-On Instruction and should not be used in Add-On Instruction definition logic or the logic does not verify.

Limits cannot be accessed inside Add-On Instruction logic.

The maximum data instance supported (which includes Inputs, Outputs, and local tags) is two megabytes. The data type size is displayed on the bottom of the Parameters and Local Tags tab in the Add-On Instruction Definition.




## Runtime Editing

Add-On Instructions can only be edited offline. If the intended functionality needs to be changed in a running controller, consider carefully if an Add-On Instruction is suitable.

## Nesting Add-On Instructions

Add-On Instructions can call other Add-On Instructions in their routines. This provides the ability to design more modular code by creating simpler instructions that can be used to build more complex functionality by nesting instructions. The instructions can be nested to seven levels deep.

Add-On Instructions cannot call other routines via a JSR instruction. You must use nested instructions if you need complex functionality consisting of multiple routines.

TIP To nest Add-On Instructions, both the nested instruction and the instruction that calls it must be of the same class type or the calling instruction will not verify. That is, standard Add-On Instructions may call only standard Add-On Instructions and safety Add-On Instructions may call only safety Add-On Instructions.

## Routines Versus Add-On Instructions

You can write your code in three basic ways: to run in-line as a main routine, to use subroutine calls, or as Add-On Instructions. The following table summarizes the advantages and disadvantages of each.

**Table 1 - Advantages and Disadvantages of Routines and Add-On Instructions**

| Aspect | Main Routine | Subroutine | Add-On Instruction |
|---|---|---|---|
| Accessibility | N/A | Within program (multiple copies, one for each program) | Anywhere in controller (single copy for the entire project) |
| Parameters | N/A | Pass by value | Pass by value via Input and Output parameters<br>or by reference via InOut parameters |
| Numeric parameters | N/A | No conversion, user must manage | Automatic data type conversion for Input and Output parameters |
| Parameters data types | N/A | Atomic, arrays, structures | Atomic for any parameter<br>Arrays and structures must be InOut parameters |
| Parameter checking | N/A | None, user must manage | Verification checks that correct type of argument has been provided for a parameter |
| Data encapsulation | N/A | All data at program or controller scope (accessible to anything) | Local data is isolated (only accessible within instruction) |
| Monitor/debug | In-line code with its data | Mixed data from multiple calls, which complicates debugging | Single calling instance data, which simplifies debugging |
| Supported programming languages | FBD, LD, SFC, ST | FBD, LD, SFC, ST | FBD, LD, ST |
| Callable from | N/A | FBD, LD, SFC, ST | FBD, LD, SFC via ST, ST |
| Protection | Locked and view only | Locked and view only | Locked and view only |
| Documentation | Routine, rung, textbox, line | Routine, rung, textbox, line | Instruction, revision information, vendor, rung, textbox, line, extended help |
| Execution performance | Fastest | JSR/SBR/RTN instructions add overhead<br>All data is copied<br>Indexed reference impact | Call is more efficient<br>InOut parameters are passed by reference, which is faster than copying data for many types<br>Parameter references are automatically offset from passed-in instruction tag location |
| Memory use | Most used | Very compact | Compact call requires more memory than a subroutine call<br>All references need an additional word |

**Table 1 - Advantages and Disadvantages of Routines and Add-On Instructions**

| Aspect | Main Routine | Subroutine | Add-On Instruction |
|---|---|---|---|
| Edit | Online/offline | Online/offline | Offline only |
| Import/export | Entire routine, including tags and instruction definitions to L5X | Entire routine, including tags and instruction definitions to L5X | Full instruction definition including routines and tags to L5X |
| Instruction signature | N/A | N/A | 32-bit signature value seals the instruction to prevent modification and provide high integrity |

## Programmatic Access to Data

Input and Output parameters and local tags are used to define an instruction-defined data type. Each parameter or local tag has a member in the data type, although local tag members are hidden from external use. Each call to an Add-On Instruction uses a tag of this data type to provide the data instance for the instruction's execution.

The parameters of an Add-On Instruction are directly accessible in the controller's programming via this instruction-defined tag within the normal tag-scoping rules.

Local tags are not accessible programmatically through this tag. This has impact on the usage of the Add-On Instruction. If a structured (including UDTs), array, or nested Add-On Instruction type is used as a local tag (not InOut parameters), then they are not programmatically available outside the Add-On Instruction definition.

TIP You can access a local tag via an alias parameter. See "Creating an Alias Parameter for a Local Tag" on page 28.

## Unavailable Instructions within Add-On Instructions

Most built-in instructions can be used within Add-On Instructions. The following instructions cannot be used.

| Unavailable Instruction | Description |
|---|---|
| BRK | Break |
| EOT | End of Transition |
| EVENT | Event Task Trigger |
| FOR | For (For/Next Loop) |
| IOT | Immediate Output |
| JSR | Jump to Subroutine |
| JXR | Jump to External Routine |
| MAOC | Motion Arm Output Cam |
| PATT | Attach to Equipment Phase |

| Unavailable Instruction | Description |
|---|---|
| PCLF | Equipment Phase Clear Failure |
| PCMD | Equipment Phase Command |
| PDET | Detach from Equipment Phase |
| POVR | Equipment Phase Override Command |
| RET | Return |
| SBR | Subroutine |
| SFP | SFC Pause |
| SFR | SFC Reset |

Safety application instructions, such as Safety Mat (SMAT), may be used in safety Add-On Instructions only. For detailed information on safety application instructions, refer to the *GuardLogix Safety Application Instruction Set Safety Reference Manual*, publication 1756-RM095.

In addition, the following instructions may be used in an Add-On Instruction, but the data instances must be passed as InOut parameters.

- ALMA (Analog Alarm)
- ALMD (Digital Alarm)
- All Motion Instructions
- MSG (Message)

## Using GSV and SSV Instructions

When using GSV and SSV instructions inside an Add-On Instruction, the following classes are supported:

- AddOnInstructionDefinition(1)(2)
- Axis
- Controller
- Controller Device
- CoordinateSystem
- CST
- DF1
- Fault Log
- Message
- MotionGroup
- Program(2)
- Routine(2)
- Redundancy
- Safety
- Serial Port
- Task(2)
- Wall Clock Time

(1) GSV-only. SSV instructions will not verify.
(2) The classes that represent programming components - Task, Program, Routine, AddOnInstructionDefinition - support only *this* as the Instance Name.

When you enter a GSV or SSV instruction, the Logix Designer application displays the object classes, object names, and attribute names for each instruction. This table lists the attributes for the AddOnInstructionDefinition class.

| Attribute Name | Data Type | Attribute Description |
|---|---|---|
| MajorRevision | DINT | Major revision number of the Add-On Instruction |
| MinorRevision | DINT | Minor revision number of the Add-On Instruction |
| Name | String | Name of the Add-On Instruction |
| RevisionExtendedText | String | Text describing the revision of the Add-On Instruction |
| Vendor | String | Vendor that created the Add-On Instruction |
| LastEditDate | LINT | Date and time stamp of the last edit to an Add-On Instruction |
| SignatureID | DINT | 32-bit instruction signature value |
| SafetySignatureID | DINT | 32-bit safety instruction signature value |

For more information on using GSV and SSV instructions, refer to the *Logix5000 Controllers General Instructions Reference Manual*, publication 1756-RM003.

## Considerations When Creating Parameters

Consider the following information when you are creating parameters.

### Passing Arguments to Parameters by Reference or by Value

The following information will help you understand the differences between passing argument tags to parameters by reference or by value.

| Aspect | By Value (Input or Output) | By Reference (InOut) |
|---|---|---|
| Value | Synchronous - the argument's value does not change during Add-On Instruction execution. | Asynchronous- the argument's value may change during Add-On Instruction execution. Any access by the instruction's logic directly reads or writes the passed tag's value. |
| Performance | Argument values are copied in and out of the parameters of the Add-On Instruction. This takes more time to execute a call to the instruction. | Parameters access argument tags directly by reference, which leads to faster execution of instruction calls. |
| Memory usage | Most amount. | Least amount. |
| Parameter data types supported | Atomic (SINT, DINT, INT, REAL, BOOL). | Atomic, arrays, and structures. |

### Selecting a Data Type for a Parameter

The Logix5000 controllers perform DINT (32 bit) and REAL (32 bit) math operations, which causes DINT data types to execute faster than other integer data types. Data conversion rules of SINT to INT to DINT are applied automatically, and can add overhead. Whenever possible, use DINT data types for the Add-On Instruction Input and Output parameters.

## Creating an Alias Parameter for a Local Tag

Alias parameters simplify connecting local tags to an Input or Output tag that is commonly used in the Add-On Instruction's application without requiring that manual code be created to make the association. Aliases can be used to define an Input or Output parameter with direct access to a local tag or its member. Changing the value of an alias parameter changes the data of the local tag or local tag member it represents and vice versa.

Alias parameters are subject to these restrictions:

- Alias parameters must be either Input or Output parameters.
- You can only create an alias parameter for a local tag or its member.
- Only one Input and one Output parameter may be mapped to the same local tag or the same member of a local tag.
- Only BOOL, SINT, INT, DINT, and REAL data types may be used.
- Alias parameters may not be constants.
- The External Access type of an alias parameter matches the External Access type of the local tag to which it is mapped.
  For information on constants and External Access, see"Data Access Control" on page 30.

## Using a Single Dimension Array as an InOut Parameter

The InOut parameter can be defined to be a single dimension array. When specifying the size of this array, consider that the user of your array can either:

- pass an array tag that is the same size as your definition.
- pass an array tag that is larger than your definition.

When developing your logic, use the Size instruction to determine the actual size of the referenced array to accommodate this flexibility.

TIP When you monitor an array InOut parameter inside of the logic routine, the parameter definition is used to determine the size of the array. For example, assume you have defined an InOut parameter to be a 10-element array of DINTs and the end user passes in an array of 100 DINTs. Then if you open the Add-On Instruction logic, select the appropriate context for that call, and monitor the array parameter, only 10 elements will be displayed.

## Determining Which Parameters to Make Visible or Required

To help you know that specific data is passed into the Add-On Instruction, you can use required parameters. A required parameter must be passed each argument for a call to the instruction to verify. In Ladder Diagram and Structured Text, this is done by specifying an argument tag for these parameters. In a Function Block Diagram, required Input and Output parameters must be wired, and InOut parameters must have an argument tag. If a required parameter does not have an argument associated, as described above, then the routine containing the call to the Add-On Instruction will not verify.

For Output parameters, making a parameter visible is useful if you do not usually need to pass the parameter value out to an argument, but you do want to display its value prominently for troubleshooting.

Required parameters are always visible, and InOut parameters are always required and visible. All Input and Output parameters, regardless of being marked as required or visible, can be programmatically accessed as a member of the instruction's tag.

**Figure 1 - Simulation Instruction in Function Block**

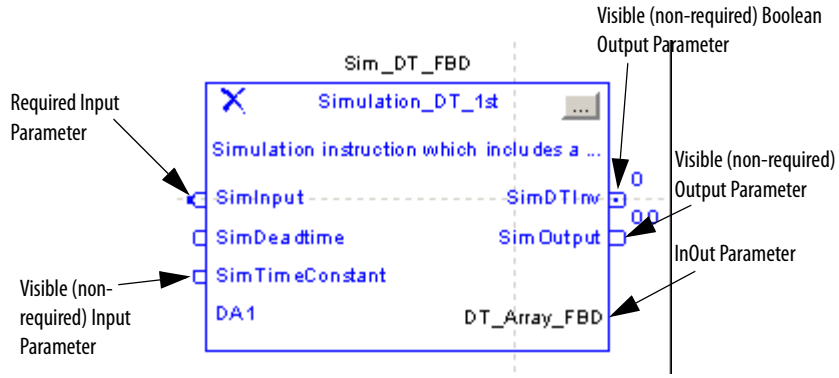
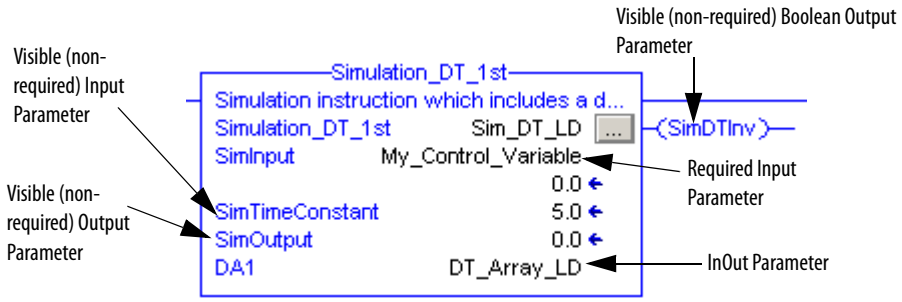



**Figure 2 - Simulation Instruction in Ladder**




-If you want the parameter's value displayed on the instruction face in Ladder, set the parameter as visible.
-An Output parameter of the BOOL tag type that is not required, but visible, will show as a status flag on the right side of the block in Ladder. This can be used for status flags like DN or ER.

This table explains the effects of the Required and Visible parameter settings on the display of the instruction.

**Table 2 - Input, Output, and InOut Required and Visible Settings**

| Parameter Type | Is the Parameter Required? | Is the Parameter Visible? | Ladder Diagram | | Function Block Diagram | | | Structured Text |
|---|---|---|---|---|---|---|---|---|
| | | | Does the Value display? | Does the Argument display? | Do You Need to Connect the Parameter? | Does the Argument display? | Can You Change the Visibility Setting Within the Function Block? | Does the Argument display? |
| Input | Y | Y | Y | Y | Y | N/A | N | Y |
| Input | N | Y | Y | N | N | N/A | Y | N |
| Input | N | N | N | N | N | N/A | Y | N |
| Output | Y | Y | Y | Y | Y | N/A | N | Y |
| Output | N | Y | Y | N | N | N/A | Y | N |
| Output | N | N | N | N | N | N/A | Y | N |
| InOut | Y | Y | N | Y | N/A | Y | N | Y |

If you have a parameter for which the user must specify a tag as its source for input or its destination as output, and you do not want this to be optional, set the parameter as required. Any required parameters are automatically set to visible.

The Visible setting is always set to visible for InOut parameters. All InOut parameters are required.

TIP When you are using your Add-On Instructions, the Visible setting may be overridden in Function Block Diagram routines if the parameter is not required or already wired. Overriding the visibility at the instruction call does not affect this definition configuration.

## Using Standard and Safety Tags

When creating a safety Add-On Instruction, follow these guidelines for standard and safety tags:

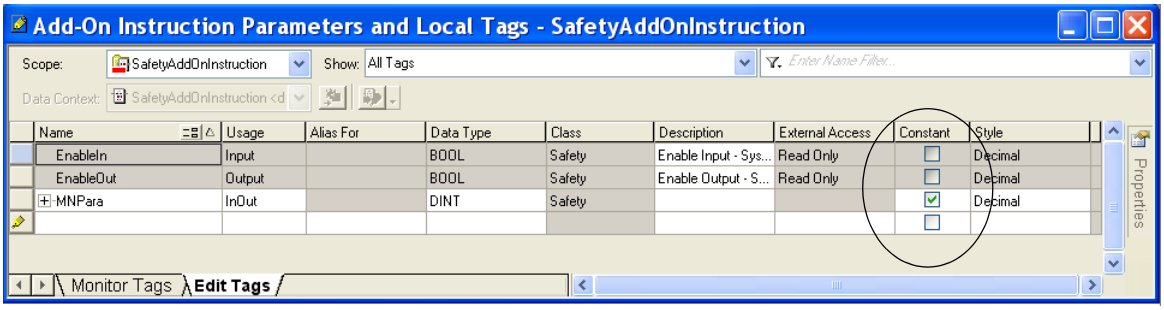
- Standard tags may not be used as Input, Output, or InOut parameters of a safety Add-On Instruction.
- Safety tags may be used as Input parameters for standard Add-On Instructions.

## Data Access Control

In RSLogix 5000 software, version 18 and later, you can prevent programmatic modification of InOut parameters by designating them as constants. You can also configure the type of access you will allow external devices, such as an HMI, to have to your tag and parameter data. You can control access to tag data changes with the Logix Designer application by configuring FactoryTalk security.

### *Constant Values*

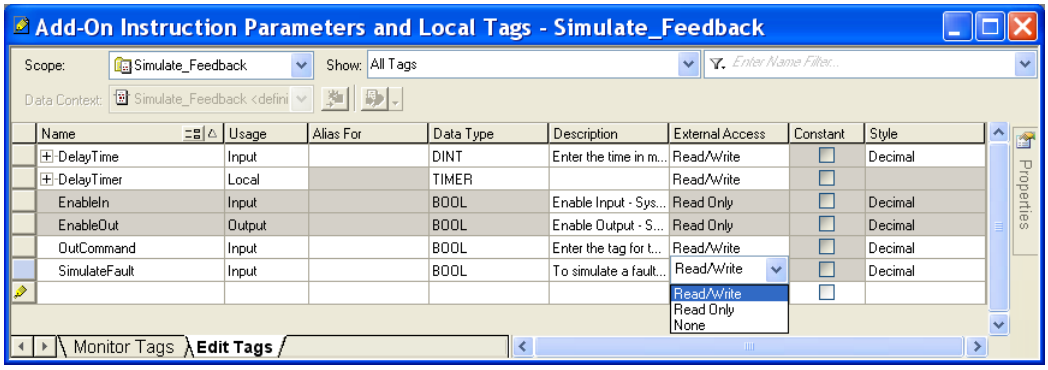
InOut parameters may be designated as constant value tags to prevent their data from being modified by controller logic. If the logic of an Add-On Instruction contains a write operation to a constant value parameter, the Add-On Instruction will not verify in the Add-On Instruction definition context.




### *External Access*

External Access defines the level of access that is allowed for external devices, such as an HMI, to see or modify tag values.

| Add-On Instruction Parameters and Tags | External Access Options |
|---|---|
| Local tag | Read/Write<br>Read Only<br>None |
| Input parameter | |
| Output parameter | |
| EnableIn parameter | Read Only |
| EnableOut parameter | |
| InOut parameter | N/A |



## Planning Your Add-On Instruction Design

Take time to plan your instruction design. Advance planning can identify issues that need to be addressed. When defining the requirements of an instruction, you are also determining the interface. Keep the following aspects in mind when defining your instruction requirements and creating your Add-On Instruction.

### Intended Behavior

- What is the purpose for creating the Add-On Instruction?
- What problem is it expected to solve?
- How is it intended to function?
- Do you need to higher level of integrity on your Add-On Instruction?

  If so, you can generate an instruction signature as a means to verify that your Add-On Instruction has not been modified.
- Do you need to use safety application instructions and certify your safety Add-On Instruction to SIL-3 integrity?

  For details on how to certify a safety Add-On Instruction, refer to the *GuardLogix Controller Systems Safety Reference Manual*, publication 1756-RM093.

### Parameters

- What data needs to be passed to the instruction?
- What information needs to be accessible outside of the instruction?
- Will alias parameters need to be defined for data from local tags that needs to be accessible from outside the Add-On Instruction?
- How will the parameters display? The order of the parameters defines the appearance of instruction.
- Which parameters should be required or visible?

### Naming Conventions

The instruction name will be used as the mnemonic for your instruction. Although the name can be up to 40 characters long, you will typically want to use shorter, more manageable names.

### Source Protection

- What type of source protection needs to be defined, if any?
- Who will have access to the source key?
- Will you need to manage source protection and an instruction signature?

Source protection can be used to provide read-only access of the Add-On Instruction or to completely lock or hide the Add-On Instruction and local tags.

Source protection must be applied prior to generating an instruction signature.

## Nesting - Reuse Instructions

- Are there other Add-On Instructions that you can reuse?
- Do you need to design your instructions to share common code?

## Local Tags

- What data is needed for your logic to execute but is not public?
- Identify local tags you might use in your instruction. Local tags are useful for items such as intermediate calculation values that you do not want to expose to users of your instruction.
- Do you want to create an alias parameter to provide outside access to a local tag?

## Programming Languages

- What language do you want to use to program your instruction?

  The primary logic of your instruction will consist of a single routine of code. Determine which programming language to use based on the use and type of application. Safety Add-On Instructions are restricted to Ladder Diagram.
- If execution time and memory usage are critical factors, refer to the *Logix5000 Controllers Execution Time and Memory Use Reference Manual*, publication 1756-RM087.

## Scan Mode Routines

- Do you need to provide Scan mode routines?

  You can optionally define the scan behavior of the instruction in different Scan modes. This lets you define unique initialization behaviors on controller startup (Program -> Run), SFC step postscan, or EnableIn False condition.
- In what language do Scan mode routines need to be written?

## Test

- How will you test the operation of your Add-On Instruction before commissioning it?
- What possible unexpected inputs could the instruction receive, and how will the instruction handle these cases?

## Help Documentation

- What information needs to be in the instruction help?

  When you are creating an instruction, you have the opportunity to enter information into various description fields. You will also need to develop information on how to use the instruction and how it operates.

# Defining Add-On Instructions

## Introduction


| Topic | Page |
|---|---|
| Creating an Add-On Instruction | 35 |
| Creating a Parameter | 37 |
| Creating Local Tags | 39 |
| Editing Parameters and Local Tags | 41 |
| Updates to Arguments Following Parameter Edits | 41 |
| Copying Parameter or Local Tag Default Values | 43 |
| Creating Logic for the Add-On Instruction | 44 |
| Defining Operation in Different Scan Modes | 45 |
| Enabling Scan Modes | 46 |
| Using the EnableIn and EnableOut Parameters | 51 |
| Changing the Class of an Add-On Instruction | 53 |
| Testing the Add-On Instruction | 53 |
| Defining Source Protection for an Add-On Instruction | 55 |
| Generating an Add-On Instruction Signature | 56 |
| Creating Instruction Help | 59 |
| Motor Starter Instruction Example | 64 |
| Simulation Instruction Example | 66 |

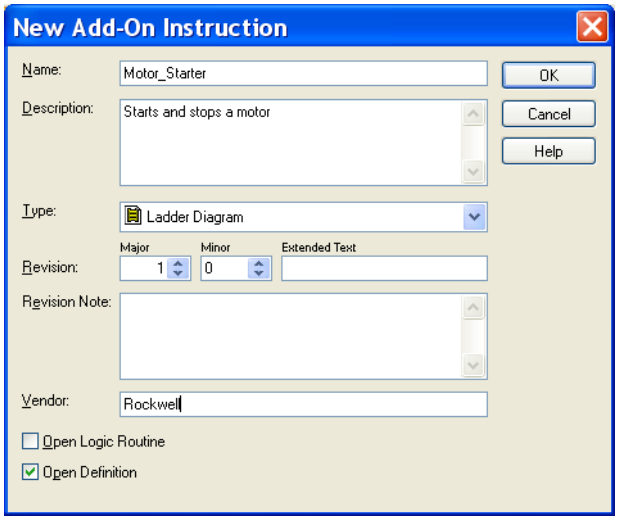

## Creating an Add-On Instruction

Follow these steps to create a new Add-On Instruction.

1. Open a new or existing project.
2. Right-click the Add-On Instructions folder in the Controller Organizer and choose **New Add-On Instruction**.

3. Type a unique name for the new instruction.

   The name can be up to 40 characters long. It must start with a letter or underscore and must contain only letters, numbers, or underscores. The name must not match the name of a built-in instruction or an existing Add-On Instruction.




4. Type a description for the new instruction, maximum 512 characters.
5. For safety projects, choose a class, either Standard or Safety.

   The Class field is available on the New Add-On Instruction dialog box for safety controller projects.

6. Choose a programming language for Add-On Instruction logic.

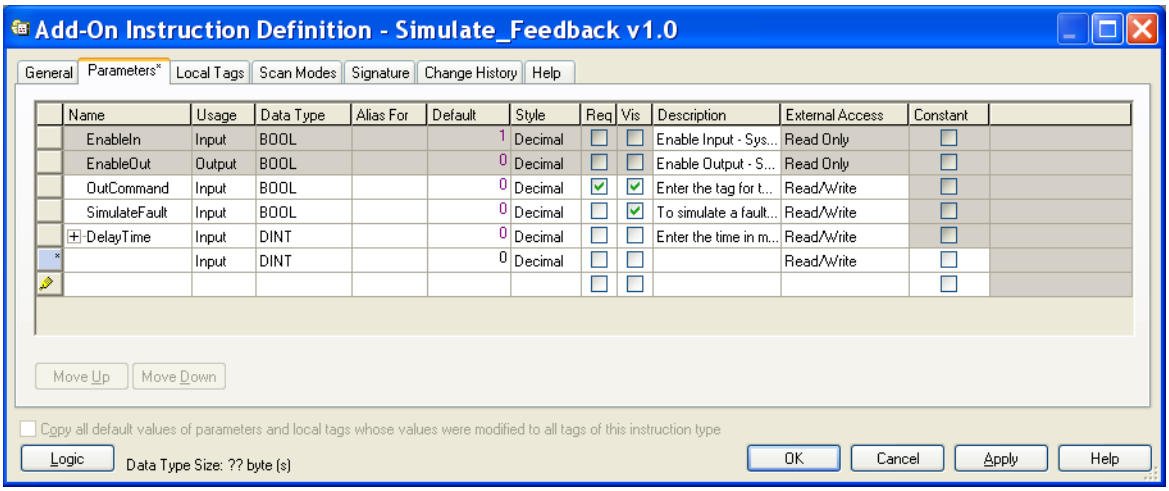
   The language Type defaults to Ladder Diagram for safety Add-On Instructions.

7. Assign a Revision level for the instruction.
8. Write a Revision note, if necessary.
9. Add information about the Vendor, if necessary.
10. Click **OK** to create the instruction.

## Creating a Parameter

Use the Add-On Instruction Definition Editor to create the parameters for your instructions. Follow these steps to define the parameters for your instruction.

1. In the Controller Organizer, right-click an Add-On Instruction and choose **Open Definition**.
2. Click the Parameters tab and enter a name for a parameter.




3. Define the Usage, based on the type of parameter: Input, Output, InOut.

   TIP An instruction with only Input parameters, except EnableOut, is treated as an input instruction in a Ladder diagram and is displayed left-justified. The EnableOut parameter is used for the rung-out condition.

4. Select a data type, with the following options based on the parameter usage:
   - An Input parameter is passed by value into the Add-On Instruction and must be a SINT, INT, DINT, REAL, or BOOL data type.
   - An Output parameter is passed by value out of the Add-On Instruction and must be a SINT, INT, DINT, REAL, or BOOL data type.
   - An InOut parameter is passed by reference into the Add-On Instruction and can be any data type including structures and array.

   TIP REAL data types are allowed for 1756-L7xS controllers.

5. If this parameter is intended as an alias for an existing local tag, click the **Alias For** menu to choose the local tag or its member.

   TIP You can also designate a parameter as an alias for a local tag by using the Tag Editor. See "Editing Parameters and Local Tags" on page 41.

6. Set the default values.

   Default values are loaded from the Add-On Instruction definition into the tag of the Add-On Instruction data type when it is created, and anytime a new Input or Output parameter is added to the Add-On Instruction.

   TIP Check the box at the bottom of the Add-On Instruction Definition Editor to 'Copy all default values of parameters and local tags whose values were modified to all tags of this instruction type' if you want to update existing invocations of the instruction to the new default values.

   For details on copying default values, see "Copying Parameter or Local Tag Default Values" on page 43.

7. Set the display style.
8. Check the box to make the parameter required or visible, as desired.

   See "Determining Which Parameters to Make Visible or Required" on page 29.

   If you decide to make the parameter required, it will also be visible.

9. Type a description, maximum 512 characters.

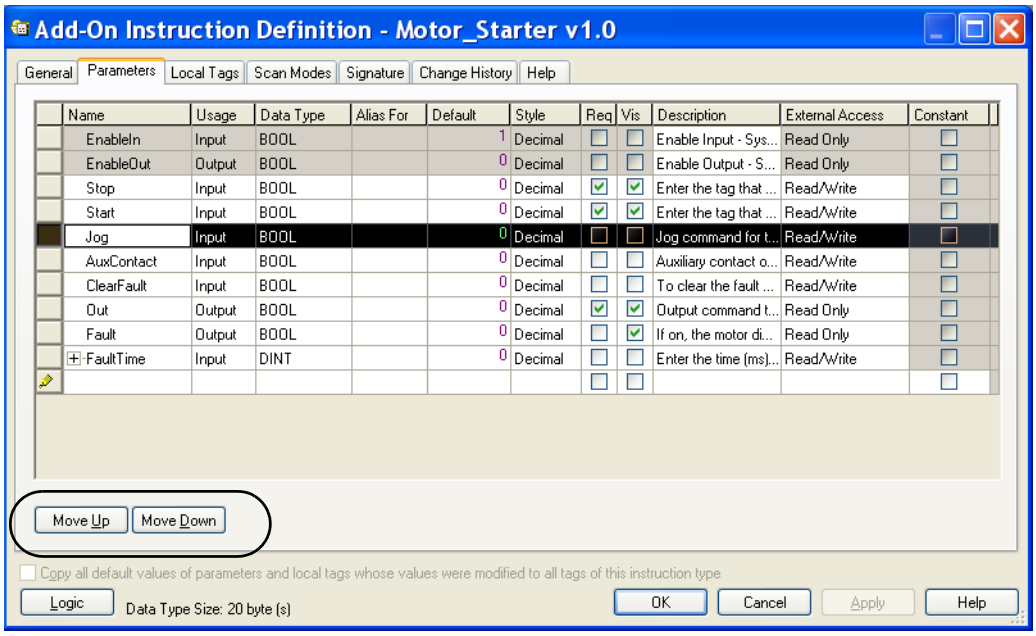
   This description appears in the instruction's help.

10. Select an External Access type for Input or Output parameters.
11. Check the **Constant** box for InOut parameters you want to designate as constant values.
12. Repeat for additional parameters.

TIP You can also create parameters by using the Tag Editor, New Parameter or Local Tag dialog box, or by right-clicking a tag name in the logic of your routine.

The order that you create the parameters is how they will appear in the data type and on the instruction face. To rearrange the order on the Parameter tab of the Add-On Instruction Definition Editor, select the parameter row and click **Move Up** or **Move Down**.

**Figure 3 - Add-On Instruction Definition Editor – Parameters Tab**




## Creating Local Tags

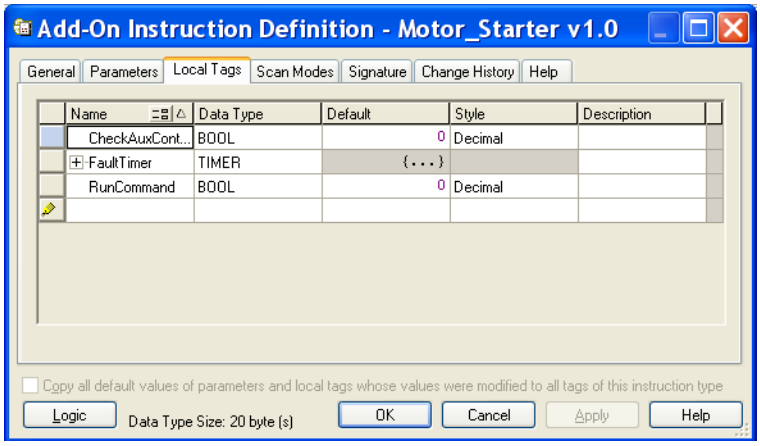
Use the Add-On Instruction Definition Editor to create the local tags for your instructions. Local tags contain data that will be used by your Add-On Instruction but that you do not want exposed to the user of your instruction. Local tags do not appear in the data structure for an Add-On Instruction because they are hidden members.

TIP You can access local tag values from an HMI by specifying the name of the local tag as a member in an Add-On Instruction type tag. For example, the Motor_Starter v1.0 instruction, shown in step 2, has a tag called *CheckAuxContact*. This tag can be referenced by an HMI via *instancetag.CheckAuxContact*, where instancetag is the tag used to call the instruction.

Follow these steps to define the local tags for your instruction.

1. In the Controller Organizer, right-click an instruction and choose **Open Definition**.

2. Click the **Local Tags** tab and type a name for a new tag and select a data type.




You cannot use these data types for local tags - ALARM_ANALOG, ALARM_DIGITAL, MESSAGE, or any Motion data types, for example Axis or MOTION_GROUP. To use these type of tags in your Add-On Instruction, define an InOut Parameter. Local tags also are limited to single dimension arrays, the same as User-Defined Data Types.

TIP Refer to the *GuardLogix Controller Systems Safety Reference Manual*, publication 1756-RM093, for a list of data types supported for safety instructions.

3. Set the default values.

Default values are loaded from the Add-On Instruction definition into the tag of the Add-On Instruction data type when it is created or any time a new tag is added to the Add-On Instruction.

TIP Check the box at the bottom of the Add-On Instruction Definition Editor to **Copy all default values of parameters and local tags whose values were modified to all tags of this instruction type** if you want to update existing invocations of the instruction to the new default values.

For details on copying default values, see "Copying Parameter or Local Tag Default Values" on page 43.

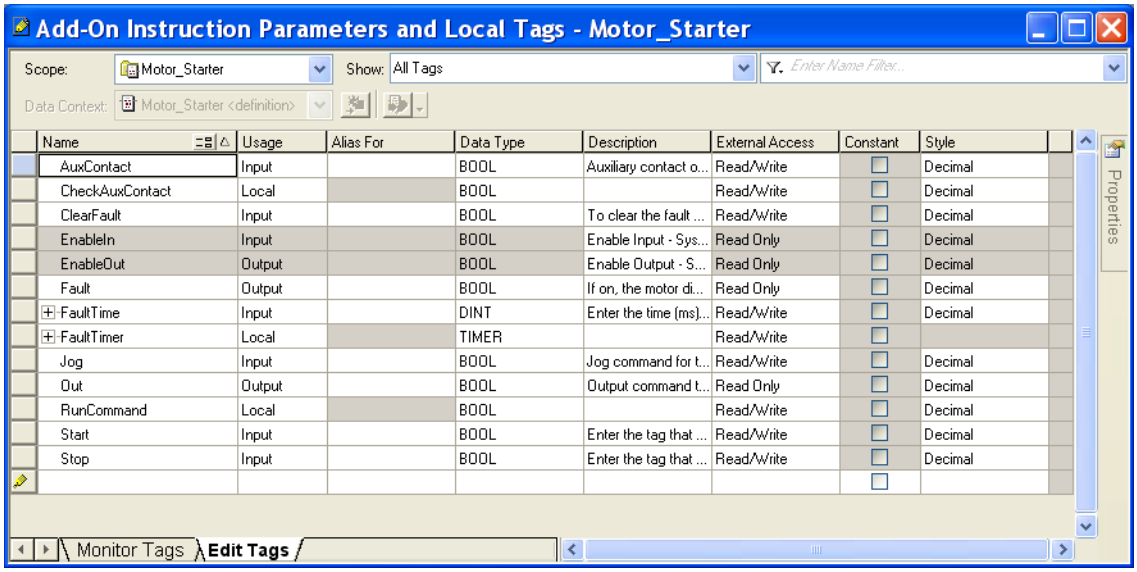
4. Set the display style.
5. Type a description, a maximum of 512 characters.
6. Repeat for additional local tags.

TIP When you create a local tag from the Local Tags tab, the External Access setting defaults to None. You can edit the External Access setting by using the Tag Editor.

See "Editing Parameters and Local Tags" on page 41.

## Editing Parameters and Local Tags

You can also add and edit parameters and local tags on the Edit Tags tab, shown below.


Add-On Instruction Parameters and Local Tags - Motor_Starter
Scope: Motor_Starter
Show: All Tags
Enter Name Filter...
Data Context: Motor_Starter <definition>
Name | Usage | Alias For | Data Type | Description | External Access | Constant | Style
AuxContact | Input | | BOOL | Auxiliary contact o... | Read/Write | | Decimal
CheckAuxContact | Local | | BOOL | | Read/Write | | Decimal
ClearFault | Input | | BOOL | To clear the fault ... | Read/Write | | Decimal
EnableIn | Input | | BOOL | Enable Input - Sys... | Read Only | | Decimal
EnableOut | Output | | BOOL | Enable Output - S... | Read Only | | Decimal
Fault | Output | | BOOL | If on, the motor di... | Read Only | | Decimal
FaultTime | Input | | DINT | Enter the time (ms)... | Read/Write | | Decimal
FaultTimer | Local | | TIMER | | Read/Write | |
Jog | Input | | BOOL | Jog command for t... | Read/Write | | Decimal
Out | Output | | BOOL | Output command t... | Read Only | | Decimal
RunCommand | Local | | BOOL | | Read/Write | | Decimal
Start | Input | | BOOL | Enter the tag that ... | Read/Write | | Decimal
Stop | Input | | BOOL | Enter the tag that ... | Read/Write | | Decimal
Monitor Tags
Edit Tags
Properties

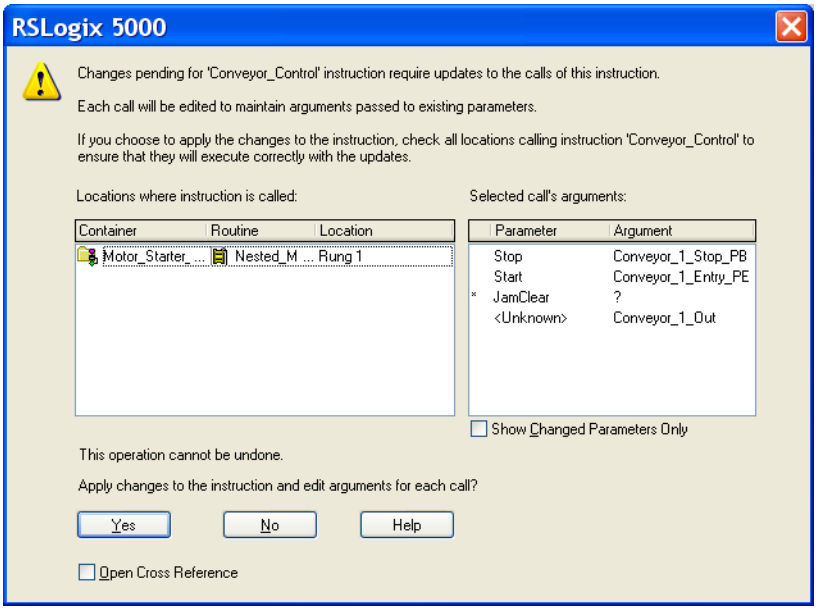
## Updates to Arguments Following Parameter Edits

If you edit an Add-On Instruction by adding, deleting, renaming, reordering, or changing the status or usage type of one or more parameters, RSLogix 5000 software, version 18 and later, automatically updates the arguments on calls to the instruction.

Source-protected routines and other source-protected Add-On Instructions that use the edited Add-On Instruction are not automatically updated if the source key is unavailable. The Add-On Instruction or routine may still verify, but the resulting operation may not be as intended.

It is your responsibility to know where Add-On Instructions are used in logic when you make edits to existing Add-On Instructions.

A confirmation dialog box shows you the impacts of the edits and lets you review the pending changes before confirming or rejecting them.




- An asterisk identifies parameters with changes pending.
- Existing arguments are reset to the parameters they were originally associated with.
- Newly added parameters are inserted with a ? in the argument field, except for Structured Text, where the field is blank.
- Unknown parameters are created for arguments where associated parameters have been deleted.

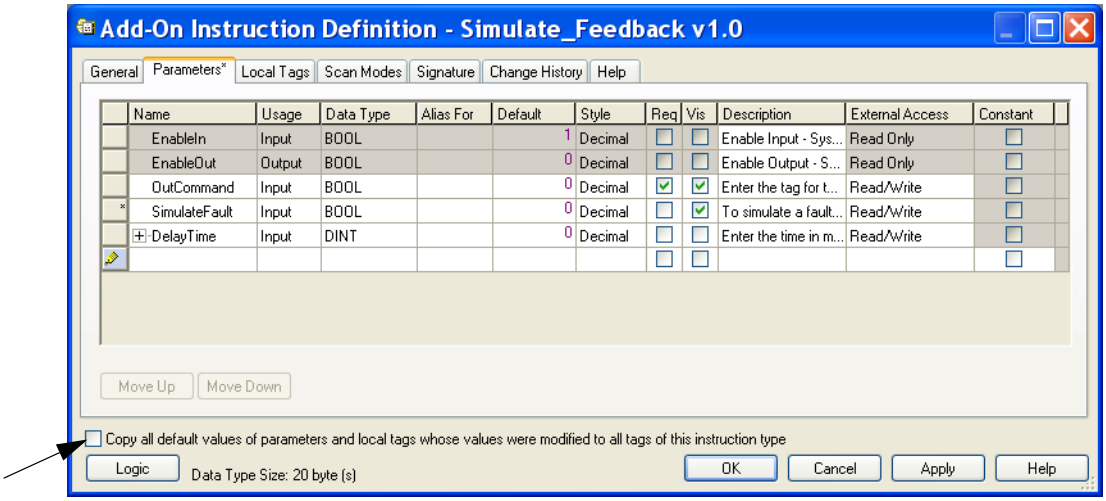
To accomplish this update, the Logix Designer application tracks the changes made to the Add-On Instruction parameters from the original instruction to the final version. In contrast, the import and paste processes compare only parameter names to associate arguments with parameters. Therefore, if two different parameters have the same name, but different operational definitions, importing or pasting may impact the behavior of the instruction.

## Copying Parameter or Local Tag Default Values

In RSLogix 5000 software, version 18 or later, you can choose to copy parameter or local tag default values to all tags of the Add-On Instruction data type or just to specific tags. You can do so only when you are offline.

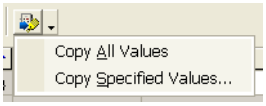
**ATTENTION:** Values cannot be modified when the instance tags are part of a source-protected Add-On Instruction or you do not have sufficient permission to make edits.

If you modify the default values of a parameter or local tag by using the Add-On Instruction Definition Editor, you can choose to copy the modified values to all of the tags of the Add-On Instruction data type by checking **Copy all default values of parameters and local tags whose values were modified to all tags of this instruction type**.

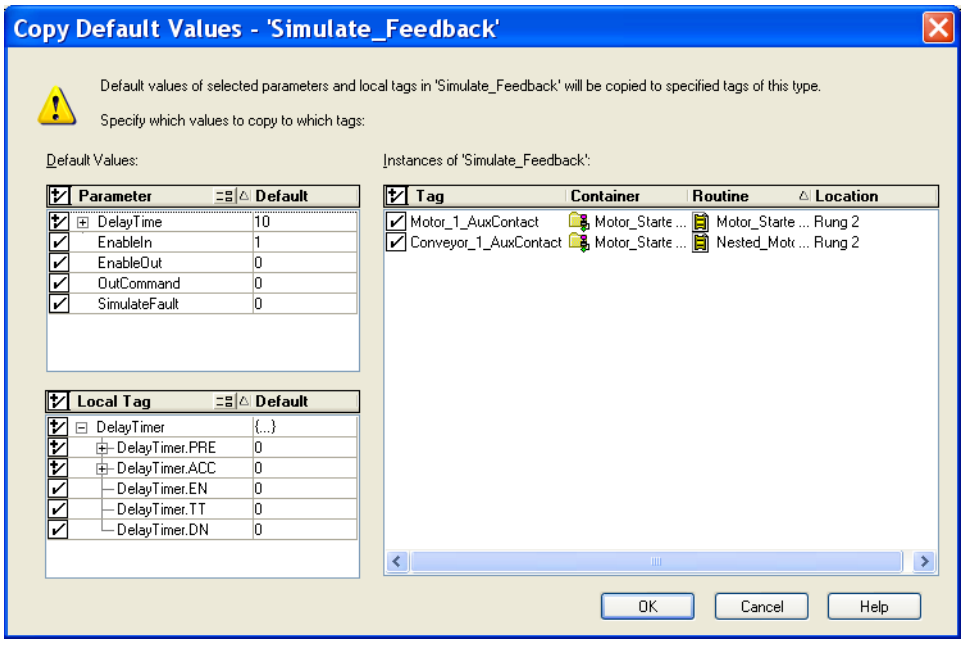



You can also click the copy default values icon to copy default values to all tags of the Add-On Instruction data type. The icon appears on the watch pane (as a context menu), data monitor, and logic editor when the Data Context is the Add-On Instruction's definition.

If you want to select which specific tags and values to copy, click the arrow of the copy default values icon and choose **Copy Specified Values**.



The Copy Default values dialog box shows the current default values for the parameters and local tags, and the instance tags where the Add-On Instruction is used or referenced.

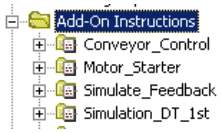



Check the boxes to select which values to copy to which tags, and click **OK**.

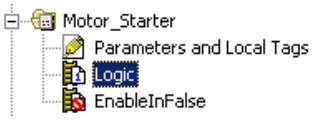
## Creating Logic for the Add-On Instruction

Follow these steps to enter logic into your project.

1. In the Controller Organizer, expand the **Add-On Instructions** folder.




2. Expand the instruction and double-click the logic routine to open.




3. Edit your logic by using the available language editors.

### Execution Considerations for Add-On Instructions

An Add-On Instruction is executed just like any other routine belonging to a particular program. Because another task can preempt a program containing an Add-On Instruction that is being executed, an Add-On Instruction may be interrupted prior to completing its execution.

In standard programs, you can use the User Interrupt Disable/Enable (UID/UIE) instructions to block a task switch if you want to be sure the Add-On Instruction executes uninterrupted before switching to another task.

TIP UID and UIE instructions are not supported in the safety task of GuardLogix projects.

### Optimizing Performance

The performance depends on the structuring, configuration, and the amount of code in an Add-On Instruction. You can pass large amounts of data through a structure by using an InOut parameter. The size of data referenced by an InOut parameter does not impact scan time and there is no difference between passing a user-defined type tag or an atomic tag because it is passed by reference.

When a rung condition is false, any calls to an Add-On Instruction are still processed even though the logic routine is not executed. The scan time can be affected when many instances of an Add-On Instruction are executed false. Be sure to provide instructions in your documentation if an Add-On Instruction can be skipped when the rung condition is false.

## Defining Operation in Different Scan Modes

To provide Add-On Instructions with the same flexibility as built-in instructions, optional Scan mode routines can be configured to fully define the behavior of the instruction. Scan mode routines do not initially exist for Add-On Instructions. You can create them depending upon the requirements of your instruction.

Like all built-in instructions in the controller, Add-On Instructions support the following four controller Scan modes.

| Scan Mode | Description |
|---|---|
| True | The instruction is scanned as the result of a true rung condition or the EnableIn parameter is set True. |
| False | The instruction is scanned as the result of a false rung condition or the EnableIn parameter is set False. Instructions in the controller may or may not have logic that executes only when that instruction is scanned false. |
| Prescan | Occurs when the controller either powers up in Run mode or transitions from Program to Run. Instructions in the controller may or may not have logic that executes only when that instruction is executed in Prescan mode. |
| Postscan(1) | Occurs as a result of an Action in an Sequential Function Chart (SFC) routine becoming inactive if SFCs are configured for Automatic Reset. Instructions in the controller may or may not have logic that executes only when that instruction is executed in Postscan mode. |

(1) Postscan mode routines cannot be created for safety Add-On Instructions because safety instructions do not support SFC.

The default behavior for executing an Add-On Instruction with no optional scan routines created may be sufficient for the intended operation of the instruction. If you do not define an optional Scan Mode, the following default behavior of an Add-On Instruction occurs.

| Scan Mode | Result |
|---|---|
| True | Executes the main logic routine of the Add-On Instruction. |
| False | Does not execute any logic for the Add-On Instruction and does not write any outputs. Input parameters are passed values. |
| Prescan | Executes the main logic routine of the Add-On Instruction in Prescan mode. Any required Input and Output parameters' values are passed. |
| Postscan | Executes the main logic routine of the Add-On Instruction in Postscan mode. |

For each Scan mode, you can define a routine that is programmed specifically for that Scan mode and can be configured to execute in that mode.

| Scan Mode | Result |
|---|---|
| True | The main logic routine for the Add-On Instruction executes (not optional). |
| False | The EnableIn False routine executes normally in place of the main logic when a scan false of the instruction occurs. Any required (or wired in FBD) Input and Output parameters' values are passed. |
| Prescan | The Prescan routine executes normally after a prescan execution of the main logic routine. Any required Input and Output parameters' values are passed. |
| Postscan | The Postscan routine executes normally after a postscan execution of the main logic routine. |

## Enabling Scan Modes

The Scan Modes tab in the Instruction Definition Editor lets you create and enable execution of the routines for the three Scan modes: Prescan, Postscan, and EnableInFalse.

### Prescan Routine

When the controller transitions from Program mode to Run mode or when the controller powers up in Run mode, all logic within the controller is executed in Prescan mode. During this scan, each instruction may initialize itself and some instructions also initialize any tags they may reference. For most instructions, Prescan mode is synonymous with scanning false. For example, an OTE instruction clears its output bit when executed during Prescan mode. For others, special initialization may be done, such as an ONS instruction setting its storage bit during Prescan mode. During Prescan mode, all instructions evaluate false so conditional logic does not execute.

The optional Prescan routine for an Add-On Instruction provides a way for an Add-On Instruction to define additional behavior for Prescan mode. When a Prescan routine is defined and enabled, the Prescan routine executes normally after the primary logic routine executes in Prescan mode. This is useful when you want to initialize tag values to some known or predefined state prior to execution.

An example would be setting a PID instruction to Manual mode with a 0% output prior to its first execution or initializing some coefficient values in your Add-On Instruction.

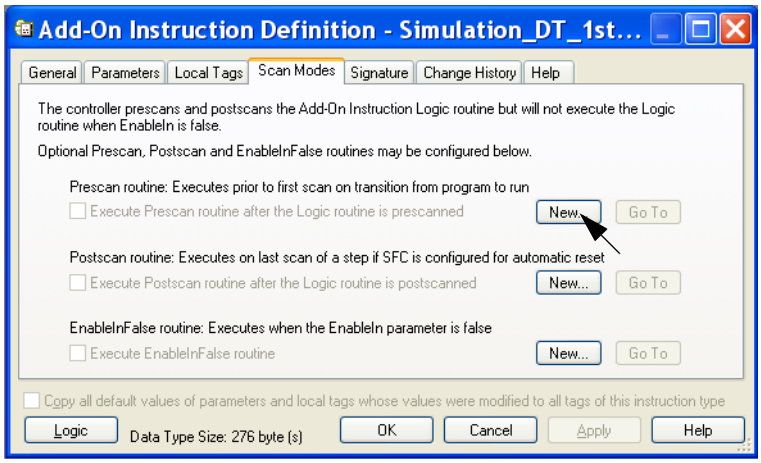
When an Add-On Instruction executes in Prescan mode, any required parameters have their data passed.

- Values are passed to Input parameters from their arguments in the instruction call.
- Values are passed out of Output parameters to their arguments defined in the instruction call.

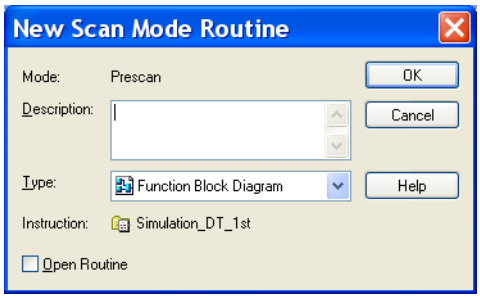
These values are passed even when the rung condition is false in Ladder Diagram or when the instruction call is in a false conditional statement in Structured Text. When Function Block Diagram routines execute, the data values are copied to all wired inputs and from all wired outputs, whether or not the parameters are required.

Follow these steps to create a Prescan routine.

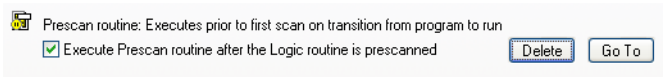
1. In the Controller Organizer, right-click an instruction and choose **Open Definition**.
2. Click the **Scan Modes** tab.
3. Click **New** for Prescan Routine.


Add-On Instruction Definition - Simulation_DT_1st...
General
Parameters
Local Tags
Scan Modes
Signature
Change History
Help
The controller prescans and postscans the Add-On Instruction Logic routine but will not execute the Logic routine when EnableIn is false.
Optional Prescan, Postscan and EnableInFalse routines may be configured below.
Prescan routine: Executes prior to first scan on transition from program to run
Execute Prescan routine after the Logic routine is prescanned
New...
Go To
Postscan routine: Executes on last scan of a step if SFC is configured for automatic reset
Execute Postscan routine after the Logic routine is postscanned
New...
Go To
EnableInFalse routine: Executes when the EnableIn parameter is false
Execute EnableInFalse routine
New...
Go To
Copy all default values of parameters and local tags whose values were modified to all tags of this instruction type
Logic
Data Type Size: 276 byte (s)
OK
Cancel
Apply
Help

4. From the Type menu on the New Scan Mode Routine dialog box, choose the type of programming language: Ladder Diagram, Function Block, or Structured Text.




5. Type a description of the Prescan behavior.
6. Click **OK** to create the routine and return to the Scan Modes tab.
7. Define if the prescan routine executes (or not) by checking or clearing **Execute Prescan routine after the logic routine is prescanned**.


Prescan routine: Executes prior to first scan on transition from program to run
Execute Prescan routine after the Logic routine is prescanned
Delete
Go To

The Prescan routine can now be edited like any other routine.

## Postscan Routine

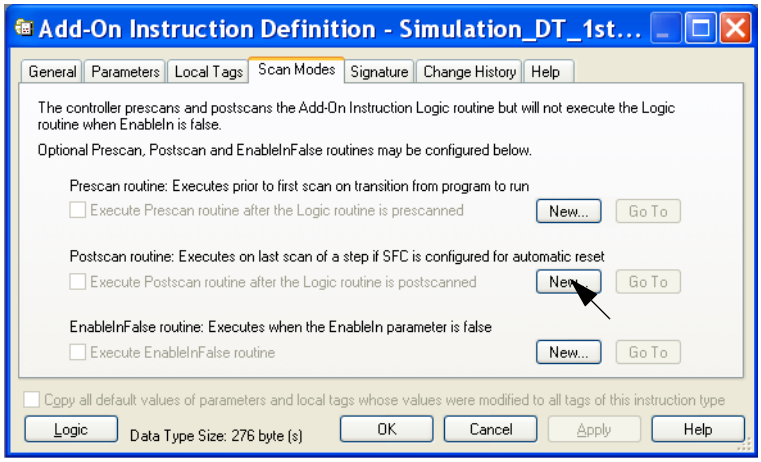
Postscan mode occurs only for logic in a Sequential Function Chart Action when the Action becomes inactive and the SFC language is configured for Automatic Reset (which is not the default option for SFC). When an SFC Action becomes inactive, then the logic in the Action is executed one more time in Postscan mode. This mode is similar to Prescan in that most instructions simply execute as if they have a false condition. It is possible for an instruction to have different behavior during Postscan mode than it has during Prescan mode.

When an Add-On Instruction is called by logic in an SFC Action or a call resides in a routine called by a JSR from an SFC Action, and the Automatic Reset option is set, the Add-On Instruction executes in Postscan mode. The primary logic routine of the Add-On Instruction executes in Postscan mode. Then, if it is defined and enabled, the Postscan routine for the Add-On Instruction executes. This could be useful in resetting internal states, status values, or de-energizing instruction outputs automatically when the action is finished.

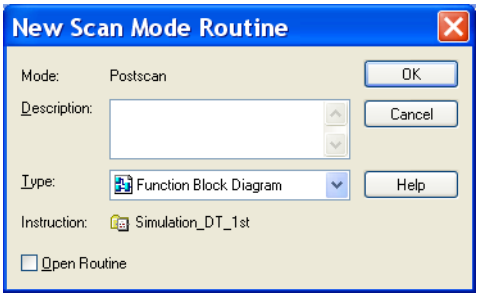
TIP Because safety Add-On Instructions cannot be called from an SFC Action, this option is not available for safety Add-On Instructions.

Follow these steps to create a postscan routine.

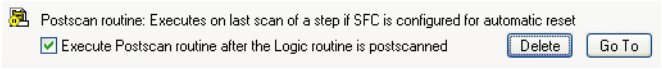
1. In the Controller Organizer, right-click an instruction and choose **Open Definition**.
2. Click the **Scan Modes** tab.
3. Click **New** for Postscan Routine.


Add-On Instruction Definition - Simulation_DT_1st...
General
Parameters
Local Tags
Scan Modes
Signature
Change History
Help
The controller prescans and postscans the Add-On Instruction Logic routine but will not execute the Logic routine when EnableIn is false.
Optional Prescan, Postscan and EnableInFalse routines may be configured below.
Prescan routine: Executes prior to first scan on transition from program to run
Execute Prescan routine after the Logic routine is prescanned
New...
Go To
Postscan routine: Executes on last scan of a step if SFC is configured for automatic reset
Execute Postscan routine after the Logic routine is postscanned
New...
Go To
EnableInFalse routine: Executes when the EnableIn parameter is false
Execute EnableInFalse routine
New...
Go To
Copy all default values of parameters and local tags whose values were modified to all tags of this instruction type
Logic
Data Type Size: 276 byte (s)
OK
Cancel
Apply
Help

4. From the Type menu on the New Scan Mode Routine dialog box, choose the type of programming language: Ladder Diagram, Function Block, or Structured Text.




5. Type a description of the postscan behavior.
6. Click **OK** to create the routine and return to the Scan Modes tab.
7. Define if the postscan routine executes (or not) by checking or clearing **Execute Postscan routine after the logic routine is postscanned**.


Postscan routine: Executes on last scan of a step if SFC is configured for automatic reset
Execute Postscan routine after the Logic routine is postscanned
Delete
Go To

The Postscan routine can now be edited like any other routine.

## EnableInFalse Routine

When defined and enabled for an Add-On Instruction, the EnableInFalse routine executes when the rung condition is false or if the EnableIn parameter of the Add-On Instruction is false (0). This is useful primarily for scan false logic, when used as an output instruction in a Ladder routine. A common use of scan false is the setting of OTEs to the de-energized state when the preceding rung conditions are false. An Add-On Instruction can use the EnableInFalse capability to let you define behavior for the False conditions.

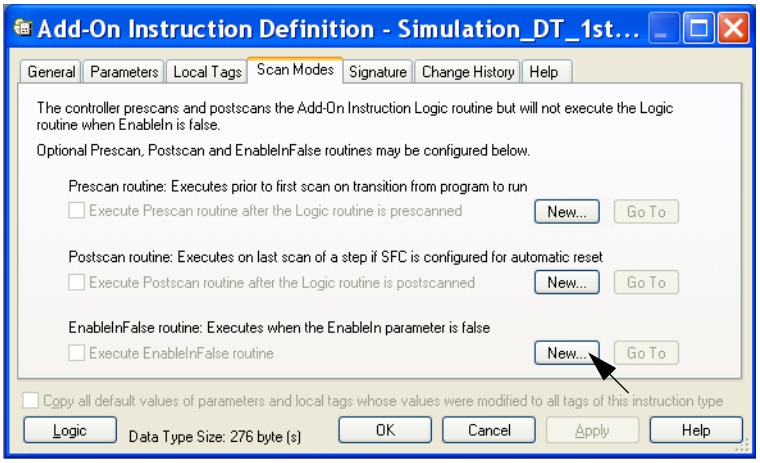
When the Add-On Instruction is executed in the false condition and has an EnableInFalse routine defined and enabled, any required parameters have their data passed.

- Values are passed to Input parameters from their arguments in the instruction call.
- Values are passed out of Output parameters from their arguments in the instruction call.

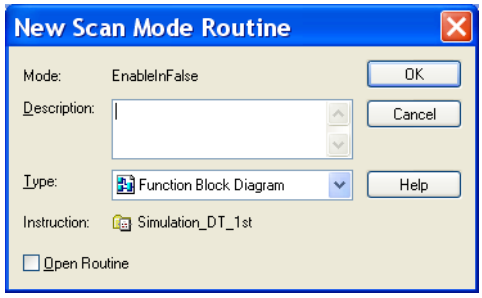
If the EnableInFalse routine is not enabled, the only action performed for the Add-On Instruction in the false condition is that the values are passed to any required Input parameters in ladder logic.

Follow these steps to create an EnableInFalse routine.

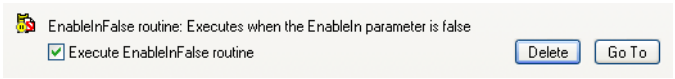
1. Right-click the instruction and choose **Open Definition**.
2. Click the **Scan Modes** tab.
3. Click **New** on EnableInFalse routine.


Add-On Instruction Definition - Simulation_DT_1st...
General
Parameters
Local Tags
Scan Modes
Signature
Change History
Help
The controller prescans and postscans the Add-On Instruction Logic routine but will not execute the Logic routine when EnableIn is false.
Optional Prescan, Postscan and EnableInFalse routines may be configured below.
Prescan routine: Executes prior to first scan on transition from program to run
Execute Prescan routine after the Logic routine is prescanned
New...
Go To
Postscan routine: Executes on last scan of a step if SFC is configured for automatic reset
Execute Postscan routine after the Logic routine is postscanned
New...
Go To
EnableInFalse routine: Executes when the EnableIn parameter is false
Execute EnableInFalse routine
New...
Go To
Copy all default values of parameters and local tags whose values were modified to all tags of this instruction type
Logic
Data Type Size: 276 byte (s)
OK
Cancel
Apply
Help

4. From the Type menu on the New Scan Mode Routine dialog box, choose the type of programming language: Ladder, Function Block, or Structured Text.




5. Type a description of the EnableInFalse behavior.
6. Click **OK** to add an EnableInFalse routine to the Add-On Instruction definition.
7. Define if EnableIn False routine executes (or not) by checking or clearing **Execute EnableInFalse routine**.




The EnableInFalse routine can now be edited like any other routine.

## Using the EnableIn and EnableOut Parameters

The EnableIn and EnableOut parameters that appear by default in every Add-On Instruction have behaviors that conform to the three language environments: Ladder Diagram, Function Block Diagram, and Structured Text.

To execute the primary logic routine in any of the language environments, the EnableIn parameter must be True (1). In general, the EnableIn parameter should not be referenced by the primary logic routine within the instruction definition. The EnableOut parameter will, by default, follow the state of the EnableIn parameter but can be overridden by user logic to force the state of this Parameter.

TIP If EnableIn is False, then EnableOut cannot be made True in an EnableIn False routine.

If the EnableIn parameter of the instruction is False (0), the logic routine is not executed and the EnableOut parameter is set False (0). If an EnableInFalse routine is included in the instruction definition and it is enabled, the EnableInFalse routine will be executed.

## EnableIn Parameter and Ladder Diagram

In the ladder diagram environment, the EnableIn parameter reflects the rung state on entry to the instruction. If the rung state preceding the instruction is True (1), the EnableIn parameter will be True and the primary logic routine of the instruction will be executed. Likewise, if the rung state preceding the instruction is False (0), the EnableIn parameter will be False and the primary logic routine will not be executed.

TIP An instruction with only Input parameters, except EnableOut, is treated as an input instruction (left-justified) in a Ladder Diagram. The EnableOut parameter is used for the rung-out condition.

## EnableIn Parameter and Function Blocks

In the function block environment, the EnableIn parameter can be manipulated by the user through its pin connection. If no connection is made, the EnableIn parameter is set True (1) when the instruction begins to execute and the primary logic routine of the instruction will be executed. If a wired connection to this parameter is False (0), the primary logic routine of the instruction will not execute. Another reference writing to the EnableIn parameter, such as a Ladder Diagram rung or a Structured Text assignment, will have no influence on the state of this parameter. Only a wired connection to this parameter's input pin can force it to be False (0).

## EnableIn Parameter and Structured Text

In the Structured Text environment, the EnableIn parameter is always set True (1) by default. The user cannot influence the state of the EnableIn parameter in a Structured Text call to the instruction. Because EnableIn is always True (1) in Structured Text, the EnableInFalse routine will never execute for an instruction call in Structured Text.

## Changing the Class of an Add-On Instruction

You can change the class of a safety Add-On Instruction so that it can be used in a standard task or standard controller. You can change the class in a safety project if the instruction does not have an instruction signature, you are offline, the application does not have a safety task signature, and is not safety-locked.

You can also change the class from standard to safety so that the Add-On Instruction can be used in the safety task.

Changing the class of an Add-On Instruction results in the same class change being applied to the routines, parameters, and local tags of the Add-On Instruction. The change does not affect nested Add-On Instructions or existing instances of the Add-On Instruction.

If any parameters or tags become unverified due to the change of class, they are identified on the Parameters and Local Tags tabs of the Add-On Instruction Editor.

If any of the restrictions for safety Add-On Instructions are violated by changing the class from standard to safety, one of the following errors is displayed and the change does not succeed:

- Routines must be of Ladder Diagram type.
- Safety Add-On Instructions do not support the Postscan routine.
- One or more parameters or local tags have an invalid data type for a safety Add-On Instruction.

You must edit the parameter, tag, or routine types before the class change can be made.

TIP If the safety controller project contains safety Add-On Instructions, you must remove them from the project or change their class to standard before changing to a standard controller type.

## Testing the Add-On Instruction

You need to test and troubleshoot the logic of an instruction to get it working.

TIP When a fault occurs in an Add-On Instruction routine, a fault log is created that contains extended information useful for troubleshooting.

## Before You Test

Before you start to test an Add-On Instruction, do the following.

1. Open a project to debug offline.

   TIP Add-On Instructions can only be created or modified when offline. You can add, delete, or modify tag arguments in calls to Add-On Instructions while editing online, but you cannot edit arguments inside the Add-On Instruction while online.

2. Add the Add-On Instruction to the project, if it is not already there.

## Testing the Flow

1. Add a call to the instruction in a routine in the open project.
2. Assign any arguments to required parameters for your call.
3. Download the project.

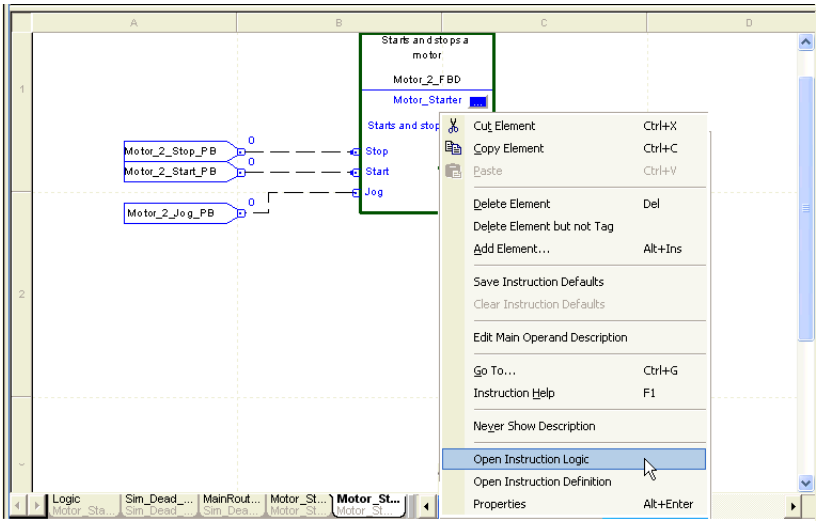
## Monitoring Logic with Data Context Views

You can simplify the online monitoring and troubleshooting of your Add-On Instruction by using Data Context views. The Data Context selector lets you select a specific call to the Add-On Instruction that defines the calling instance and arguments whose values are used to visualize the logic for the Add-On Instruction.

TIP When troubleshooting an Add-On Instruction, use a non-arrayed instance tag for the call to the instruction. This lets you monitor and troubleshoot the instruction's logic routine with a data context. Variable indexed arrays cannot be used to monitor the logic inside an Add-On Instruction.

Follow these steps to monitor the logic.

1. Go into Run mode.
2. Right-click the instruction call and choose **Open Instruction Logic**.




The logic routine opens with animated logic for the specific calling instance.

## Verifying Individual Scan Modes

The most straightforward method to verify Scan mode operation is to execute the instruction first with the Scan mode routine disabled, then again with it enabled. Then you can determine whether the Scan mode routine performed as expected.

| Instruction | Description |
|---|---|
| True | This is simply the execution of the main logic routine. |
| False | In a ladder logic target routine, this entails placing an XIC before an instance of the instruction and evaluating instruction results when the XIC is false.<br>In a Function Block target routine, this entails executing an instance of the instruction with the EnableIn parameter set to zero (0). |
| Prescan | Place the controller in Program mode, then place it in Run mode. |
| Postscan | With the controller configured for SFC Automatic Reset, place an instance of the instruction into the Action of an SFC. Run the SFC such that this Action is executed and the SFC proceeds beyond the step that is associated with this Action. |

# Defining Source Protection for an Add-On Instruction

You can apply source protection to your Add-On Instruction to protect your intellectual property or prevent unintended edits of a validated source.

With source protection you can limit a user of your Add-On Instruction to read-only access or prevent access to the internal logic or local tags used by the instruction. You can protect the use and modification of your instructions with a source key file when you distribute them.

You have two options when using source protection.

- Source Protected

  Users without the source key cannot view any routine or local tags, or make any edits to the Add-On Instruction. Choose this option if you want to protect the source definition of an Add-On Instruction from the view of a user. This may be due to the proprietary nature of the algorithms or for strict revision control.

- Source Protected with Viewable Option

  Users without the source key can view all components of the Add-On Instruction including its logic and local tags, but are prevented from making any edits to the instruction.

TIP Apply source protection before generating an instruction signature for your Add-On Instruction definition. You will need the source key to create a signature history entry.

When source protection is enabled, you can still copy the instruction signature or signature history, if they exist.

### Enable the Source Protection Feature

If source protection is unavailable and not listed in your menus, you can enable it by using the RS5KSrcPtc.exe tool on your installation CD.

For procedures on how to use the Source Protection feature, refer to the *Logix5000 Controllers Security Programming manual*, publication 1756-PM016.

## Generating an Add-On Instruction Signature

The Signature tab of the Add-On Instruction Definition Editor lets you manage the instruction signature, create signature history entries, and view the safety instruction signature, if it exists. Instruction signatures are applied to the definition of the Add-On Instruction. All instances of that Add-On Instruction are sealed when the signature is applied.

### Generating, Removing, or Copying an Instruction Signature

On the Signature tab of the Add-On Instruction Definition Editor, click **Generate** to create an instruction signature or **Remove** to delete the instruction signature.

You must be offline to generate or remove an instruction signature. Both actions change the Last Edited Date.

| IMPORTANT | If you remove an instruction signature when the Add-On Instruction also has a safety instruction signature, the safety instruction signature is also deleted. |
|---|---|

You can click **Copy** to copy the instruction signature and the safety instruction signature, if it exists, to the clipboard to facilitate record-keeping.

| IMPORTANT | If an invalid instruction signature is detected during verification, an error message indicates that the signature is invalid. You must remove the instruction signature, review the Add-On Instruction, and generate a new instruction signature. |
|---|---|

## Creating a Signature History Entry

The signature history provides a record of signatures for future reference. A signature history entry consists of the name of the user, the instruction signature, the timestamp value, and a user-defined description. You can only create a signature history if an instruction signature exists and you are offline. Creating a signature history changes the Last Edited Date, which becomes the timestamp shown in the history entry. Up to six history entries may be stored.

Follow these steps to create a signature history entry.

1. On the Signature tab of the Add-On Instruction Definition Editor, click **Add to History**.
2. Type a description, up to 512 characters long, for the entry.
3. Click **OK**.

| TIP | To facilitate record-keeping, you can copy the entire signature history to the clipboard by selecting all the rows in the signature history and choosing **Copy** from the Edit menu. The data is copied in tab separated value (TSV) format. |
|---|---|

To delete the signature history, click **Clear Signature History while the instruction does not have an instruction signature.** You must be offline to delete the Signature History.

## Generating a Safety Instruction Signature

When a sealed safety Add-On Instruction is downloaded for the first time, a SIL 3 safety instruction signature is automatically generated. Once created, the safety instruction signature is compared at every download.

If the Logix Designer application detects an invalid safety instruction signature value, it generates a new safety instruction signature value in the offline project and displays a warning indicating that the safety instruction signature was changed. The safety instruction signature is deleted if the instruction signature is removed.

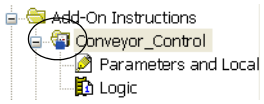
IMPORTANT

After testing the safety Add-On Instruction and verifying its functionality, you must record the instruction signature, the safety instruction signature and the timestamp value. Recording these values will help you determine if the instruction functionality has changed.

Refer to the *GuardLogix Controller Systems Safety Reference manual*, publication 1756-RM093, for details on safety application requirements.

## Viewing and Printing the Instruction Signature

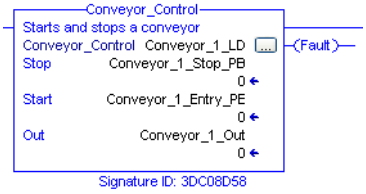
When the instruction signature has been generated, the Logix Designer application displays the instruction with the blue seal icon in the Controller Organizer, on the Add-On Instruction title bar, and in the Logic Editor.



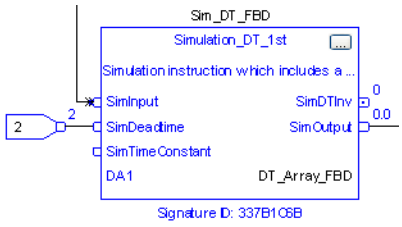

TIP

When an instruction is sealed, the instruction signature is displayed on the faceplate of the instruction in the Ladder Diagram Editor and the Function Block Diagram Editor.

**Ladder Diagram**




**Function Block Diagram**




You can turn off the display of the instruction signature in the Workstation Options dialog box of the Logix Designer application.

You can also view the instruction signature and the safety instruction signature on the Quick View pane of the Controller Organizer and on the Signature tab of the Instruction Definition Editor dialog box.

The Add-On Instruction name, revision, instruction signature, safety instruction signature, and timestamp are printed on the Add-On Instruction Signature Listing report. You can also choose to include the instruction signature, safety instruction signature, and signature history on the Add-On Instruction report by clicking **Print Options** on the Generate Report dialog box.

## Creating Instruction Help

Custom instruction help is generated automatically as you are creating your Add-On Instructions. The Logix Designer application automatically builds help for your Add-On Instructions by using the instruction's description, revision note, and parameter descriptions. By creating meaningful descriptions, you can help the users of your instruction.

In addition, you can add your own custom text to the help by using the extended description field. You can provide additional help documentation by entering it on the Help tab of the Add-On Instruction Definition Editor. The instruction help is available in the instruction browser and from any call to the instruction in a language editor by pressing **F1**.

### Writing Clear Descriptions

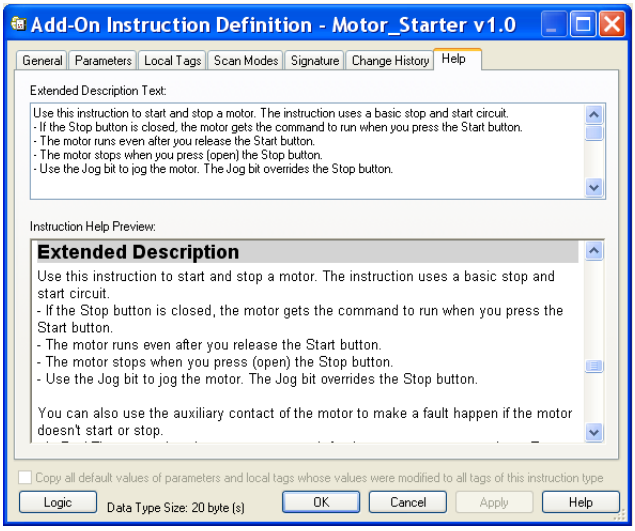
When writing your descriptions keep the following in mind:

- Use short sentences and simple language.

- Be brief and direct when you write.
- Include simple examples.
- Proofread your entries.

This is an example of the Extended Description Text field in the Help tab of the Add-On Instruction Definition Editor. This area lets you create directions on how to use and troubleshoot your instruction. The Instruction Help Preview window shows how your text will look as generated instruction help.

TIP When you are entering your text into the Extended Description Text field, you can use returns and tabs in the field to format the text, and if you copy and paste text into the field, tabs are preserved.


Add-On Instruction Definition - Motor_Starter v1.0
General
Parameters
Local Tags
Scan Modes
Signature
Change History
Help
Extended Description Text:
Use this instruction to start and stop a motor. The instruction uses a basic stop and start circuit.
- If the Stop button is closed, the motor gets the command to run when you press the Start button.
- The motor runs even after you release the Start button.
- The motor stops when you press (open) the Stop button.
- Use the Jog bit to jog the motor. The Jog bit overrides the Stop button.
Instruction Help Preview:
Extended Description
Use this instruction to start and stop a motor. The instruction uses a basic stop and start circuit.
- If the Stop button is closed, the motor gets the command to run when you press the Start button.
- The motor runs even after you release the Start button.
- The motor stops when you press (open) the Stop button.
- Use the Jog bit to jog the motor. The Jog bit overrides the Stop button.
You can also use the auxiliary contact of the motor to make a fault happen if the motor doesn't start or stop.
Copy all default values of parameters and local tags whose values were modified to all tags of this instruction type
Logic
Data Type Size: 20 byte (s)
OK
Cancel
Apply
Help

## Documenting an Add-On Instruction

Follow these steps to create custom help for an instruction.

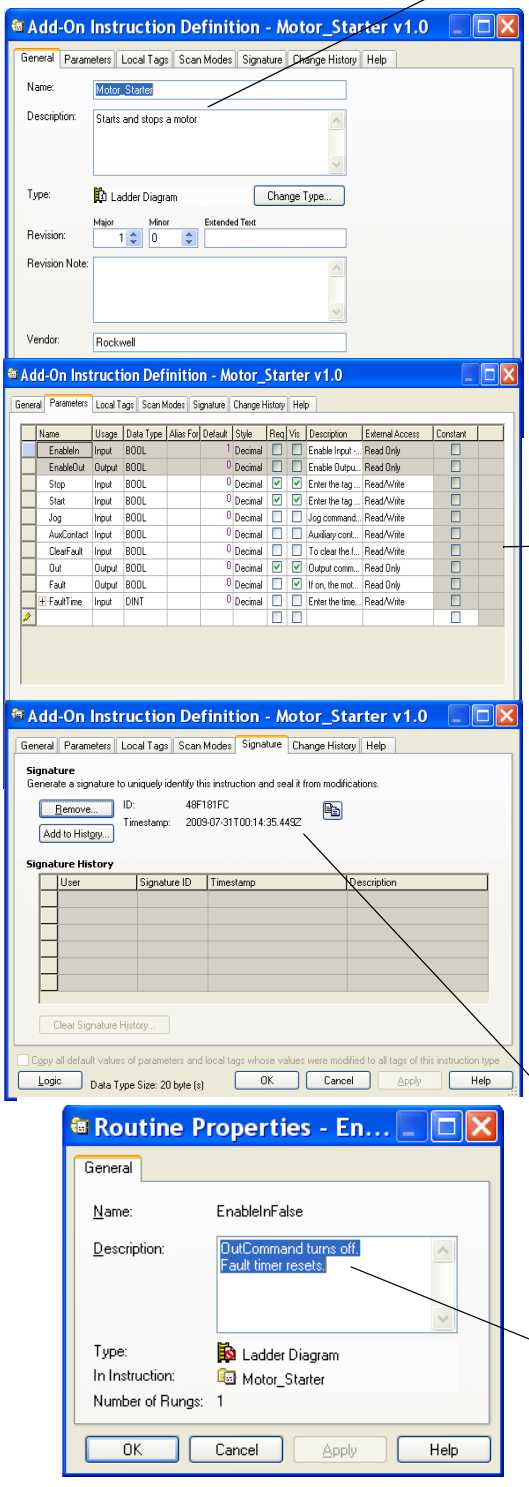
1. Right-click an Add-On Instruction and choose **Open Definition**.
2. On the General tab, enter a description and a revision note for the Add-On Instruction to explain the purpose of the instruction.
3. Click the **Parameters** tab and enter a meaningful description for each Parameter.
4. Right-click each routine located below the Add-On Instruction in the Controller Organizer and choose **Properties**.
5. Enter a description for execution of each routine.
   a. For the logic routine, describe execution of the instruction when EnableIn is true.
   b. For the EnableInFalse routine (if one exists), describe actions that will take place when EnableIn is false, such as any outputs that get cleared.
   c. For the Prescan routine (if one exists), briefly describe actions that will take place during the Prescan routine, such as initialization of any parameters.
   d. For the Postscan routine (if one exists), briefly describe actions that will take place during the Postscan routine, such as initialization of any parameters resetting any internal state of the instruction.
6. Click the **Help** tab of the Add-On Instruction Definition Editor and enter additional information in the Extended Description field.

   The extended description can include the following information:

   - Additional parameter information
   - Description of how the instruction executes
   - Change history notes

7. Review the Help format in the preview window.

This is an example of the Logix Designer application generated help for the instruction. This information is gathered from the definition descriptions that you complete when defining an instruction.




## Motor_Starter v1.0

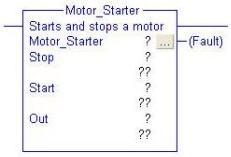
**Rockwell**

*[Contact the Add-On Instruction developer for questions or problems with this instruction]*

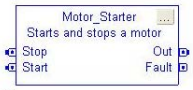
Starts and stops a motor

### Available Languages

**Relay Ladder**




**Function Block**




**Structured Text**

```
Motor_Starter(Motor_Starter,Stop,Start,Out);
```

### Parameters

| Required | Name | Data Type | Usage | Description |
|---|---|---|---|---|
| x | Motor_Starter | Motor_Starter | InOut | |
| | EnableIn | BOOL | Input | |
| | EnableOut | BOOL | Output | |
| x | Stop | BOOL | Input | Enter the tag that gives the stop command for the motor. |
| x | Start | BOOL | Input | Enter the tag that gives the start command for the motor. |
| | Jog | BOOL | Input | Jog command for the motor.<br>To jog the motor, turn on this bit.<br>To stop the jog, turn off this bit. |
| | AuxContact | BOOL | Input | Auxiliary contact of the motor.<br>Make sure you set the FaultTime.<br>Otherwise, this input doesn't do anything. |
| | ClearFault | BOOL | Input | To clear the fault of the motor, turn on this bit. |
| x | Out | BOOL | Output | Output command to the motor starter.<br>If on, the motor starts.<br>If off, the motor stops. |
| | Fault | BOOL | Output | If on, the motor didn't start or stop. |
| | FaultTime | DINT | Input | Enter the time (ms) to wait for the auxiliary contact to open or close. The Fault bit turns on when that time is up. |

### Extended Description

Use this instruction to start and stop a motor. The instruction uses a basic stop and start circuit.
- If the Stop button is closed, the motor gets the command to run when you press the Start button.
- The motor runs even after you release the Start button.
- The motor stops when you press (open) the Stop button.
- Use the Jog bit to jog the motor. The Jog bit overrides the Stop button.

You can also use the auxiliary contact of the motor to make a fault happen if the motor doesn't start or stop.
- In FaultTime, enter how long you want to wait for the contact to open or close. Enter the time in milliseconds.
- The Fault bit turns on if the contact doesn't show that the motor started or stopped within the FaultTime.
-You must set FaultTime greater than 0 to use the auxiliary contact. Otherwise the instruction doesn't use the value of the auxiliary contact.
- To clear the Fault bit, turn on the FaultClear bit.

The instruction doesn't let you enter tags for the Jog, AuxContact, and FaultClear bits in the LD and ST programming languages. You must write code to turn those bits on and off. For example:
- In LD, use XIC and OTE instructions to read the value of the auxiliary contact tag and write it to the AuxContact bit.
- In ST, use an assignment (:=) to set the AuxContact bit equal to the value of the auxiliary contact tag.

### Signature

**ID:** 48F181FC
**Timestamp:** 2009-07-31T00:14:35.449Z
**Signature History:**

| User | ID | Timestamp | Description |
|---|---|---|---|
| NA\selnis | 48F181FC | 2009-07-31T00:14:35.449Z | signature for high-integrity |

### Execution

*[see Add-On Instruction Scan Modes online Help for more information]*

| Condition | Description |
|---|---|
| EnableIn is true | OutCommand turns on when Stop and Start are on.<br>OutCommand turns off when Stop turns off. |
| EnableIn is false | OutCommand turns off.<br>Fault timer resets. |

### Revision v1.0 Notes

## Project Documentation

With RSLogix 5000 software, version 17 and later, you have the option to display project documentation, such as tag descriptions and rung comments in any supported localized language. You can store project documentation for multiple languages in a single project file rather than in language-specific project files. You define all the localized languages that the project will support and set the current, default, and optional custom localized language. The software uses the default language if the current language's content is blank for a particular component of the project. However, you can use a custom language to tailor documentation to a specific type of project file user.

Enter the localized descriptions in your project, either when programming in that language or by using the import/export utility to translate the documentation offline and then import it back into the project. Once you enable project documentation in application, you can dynamically switch between languages as you use the software.

Project documentation that supports multiple translations includes these variables:

- Component descriptions in tags, routines, programs, equipment phases, user-defined data types, and Add-On Instructions
- Engineering units and state identifiers added to tags, user-defined data types, or Add-On Instructions
- Trends
- Controllers
- Alarm Messages (in configuration of ALARM_ANALOG and ALARM_DIGITAL tags)
- Tasks
- Property descriptions for module in the Controller Organizer
- Rung comments, SFC text boxes, and FBD text boxes

If you want to allow project documentation on an Add-On Instruction that is sealed with an instruction signature, you must enter the localized documentation into your Add-On Instruction before generating the signature. Because the signature history is created after the instruction signature is generated, the signature history is not translatable. If the translated information already exists when you generate the Add-On Instruction signature, you can switch the language while keeping the signature intact because the switch does not alter the instruction definition, it only changes the language that is displayed.

For more information on enabling a project to support multiple translations of project documentation, refer to the online help.

## Motor Starter Instruction Example

The Motor_Starter Add-On Instruction starts and stops a motor.

If the stop pushbutton is closed and the start pushbutton is pressed then:

- the motor gets the command to run.
- the instruction seals in the command, so the motor keeps running even after you release the start pushbutton.

If the stop pushbutton is pressed (opened), then the motor stops.

**Figure 4 - Motor Starter Example Definition Editor General Tab**

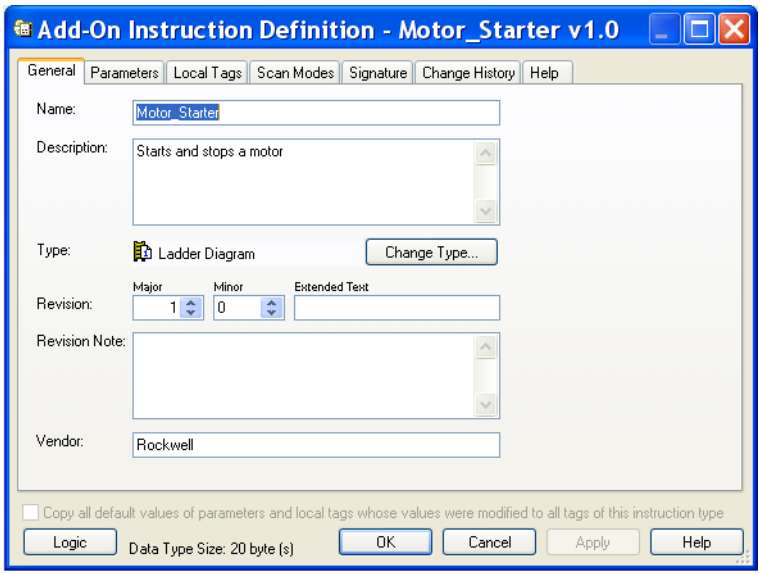



**Figure 5 - Motor Starter Example Definition Editor Parameter Tab**

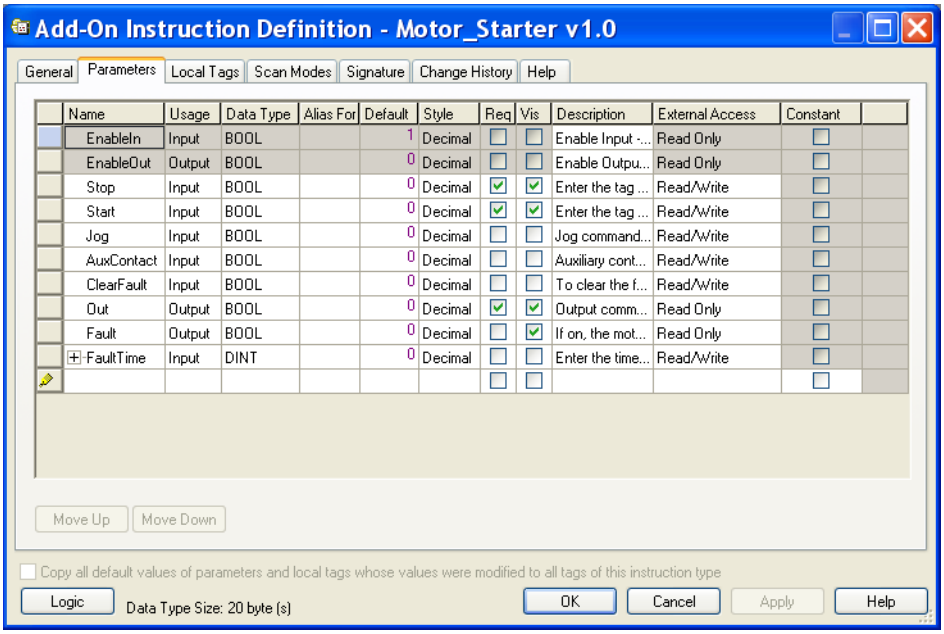
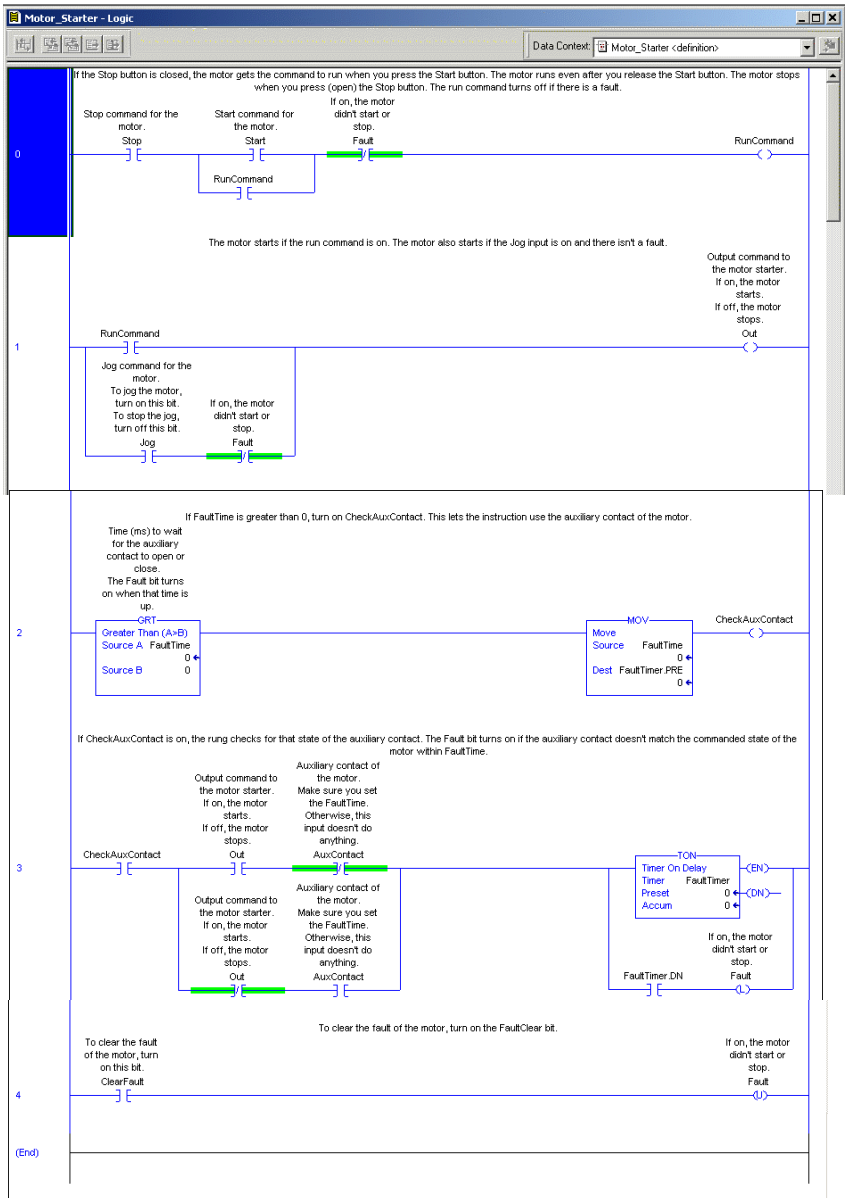


**Figure 6 - Motor Starter Example Ladder Logic**


Motor_Starter - Logic
Data Context: Motor_Starter <definition>
If the Stop button is closed, the motor gets the command to run when you press the Start button. The motor runs even after you release the Start button. The motor stops when you press (open) the Stop button. The run command turns off if there is a fault.
Stop command for the motor.
Stop
Start command for the motor.
Start
If on, the motor didn't start or stop.
Fault
RunCommand
RunCommand
The motor starts if the run command is on. The motor also starts if the Jog input is on and there isn't a fault.
Output command to the motor starter. If on, the motor starts. If off, the motor stops.
Out
RunCommand
Jog command for the motor. To jog the motor, turn on this bit. To stop the jog, turn off this bit.
Jog
If on, the motor didn't start or stop.
Fault
If FaultTime is greater than 0, turn on CheckAuxContact. This lets the instruction use the auxiliary contact of the motor.
Time (ms) to wait for the auxiliary contact to open or close. The Fault bit turns on when that time is up.
GRT
Greater Than (A>B)
Source A FaultTime
0
Source B 0
MOV
Move
Source FaultTime
0
Dest FaultTimer.PRE
0
CheckAuxContact
If CheckAuxContact is on, the rung checks for that state of the auxiliary contact. The Fault bit turns on if the auxiliary contact doesn't match the commanded state of the motor within FaultTime.
Output command to the motor starter. If on, the motor starts. If off, the motor stops.
Out
Auxiliary contact of the motor. Make sure you set the FaultTime. Otherwise, this input doesn't do anything.
AuxContact
CheckAuxContact
TON
Timer On Delay
Timer FaultTimer
Preset 0
Accum 0
EN
DN
Output command to the motor starter. If on, the motor starts. If off, the motor stops.
Out
Auxiliary contact of the motor. Make sure you set the FaultTime. Otherwise, this input doesn't do anything.
AuxContact
If on, the motor didn't start or stop.
FaultTimer.DN
Fault
To clear the fault of the motor, turn on the FaultClear bit.
To clear the fault of the motor, turn on this bit.
ClearFault
If on, the motor didn't start or stop.
Fault
(End)

The following diagrams show the Motor Starter instruction called in three different programming languages.

**Figure 7 - Motor Starter Ladder Diagram**

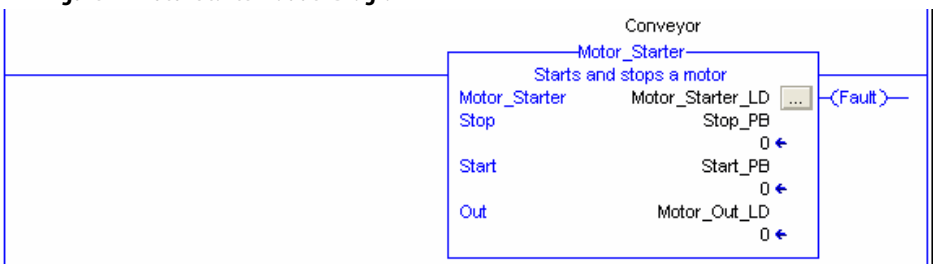


**Figure 8 - Motor Starter Function Block Diagram**

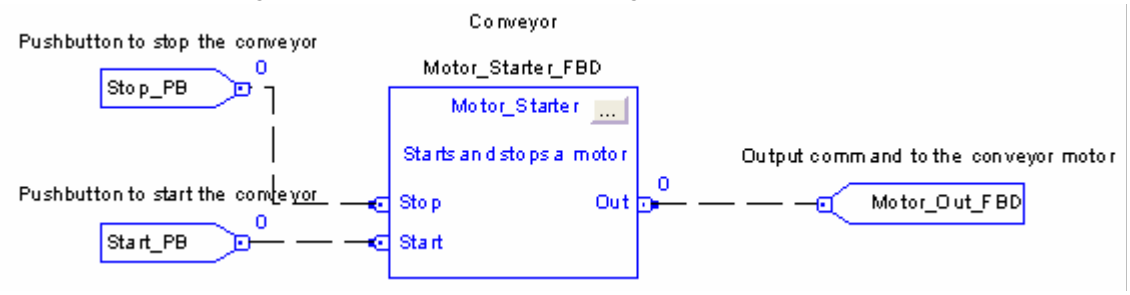



**Figure 9 - Motor Starter Structured Text**

```
Motor_Starter(Motor_Starter_ST, Stop_PB, Start_PB, Motor_Out_ST);
```

## Simulation Instruction Example

The Simulation_DT_1st Add-On Instruction adds a dead time and a first-order lag to an input variable.

**Figure 10 - Simulation Example Definition Editor General Tab**

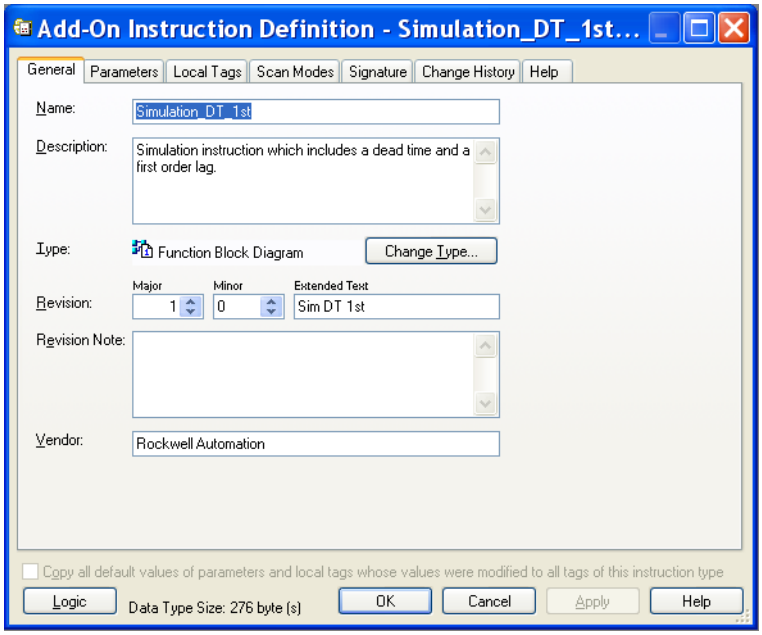


**Figure 11 - Simulation Example Definition Editor Parameter Tab**

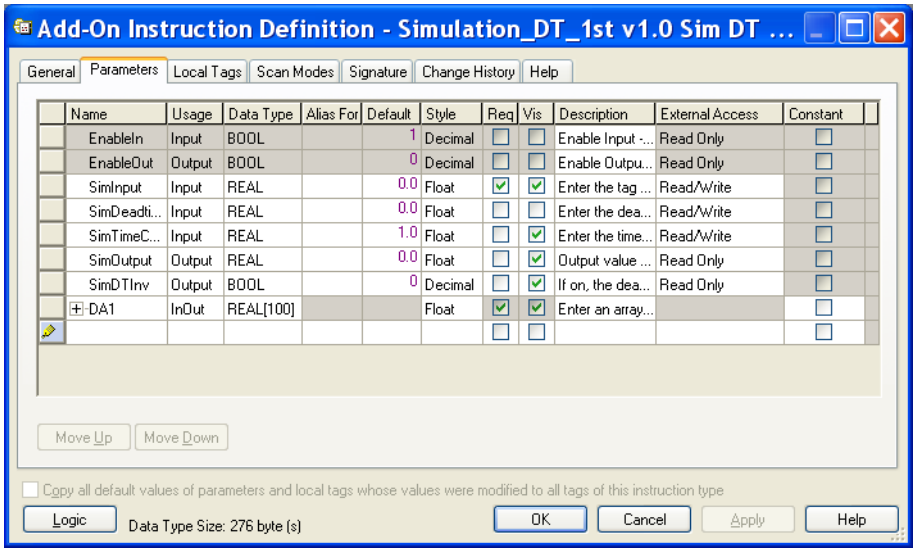
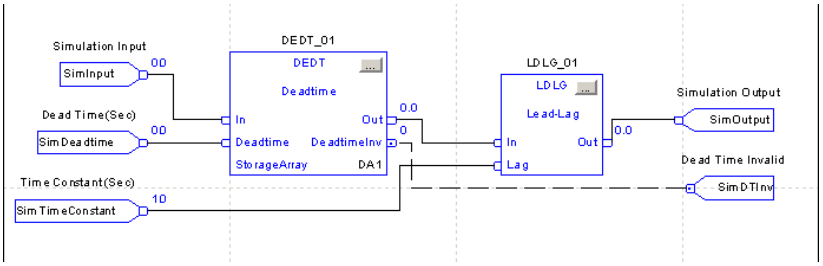


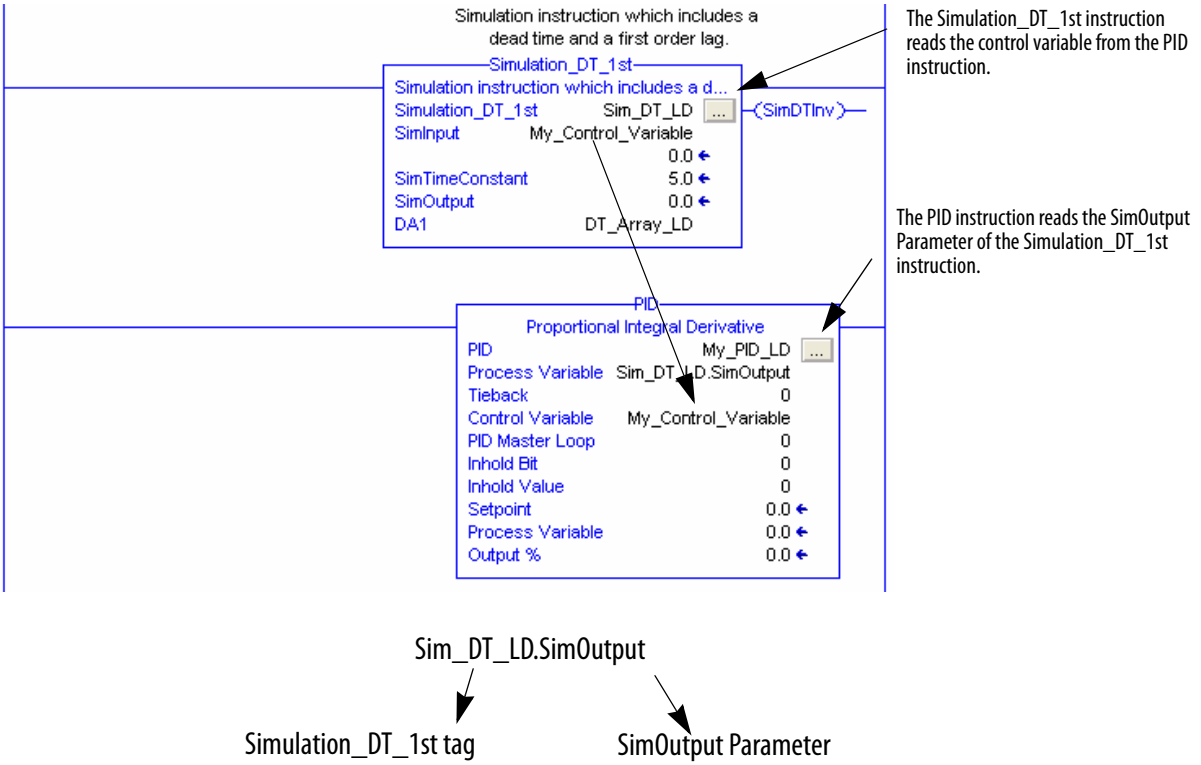

**Figure 12 - Simulation Example Logic**



## Ladder Diagram Configuration

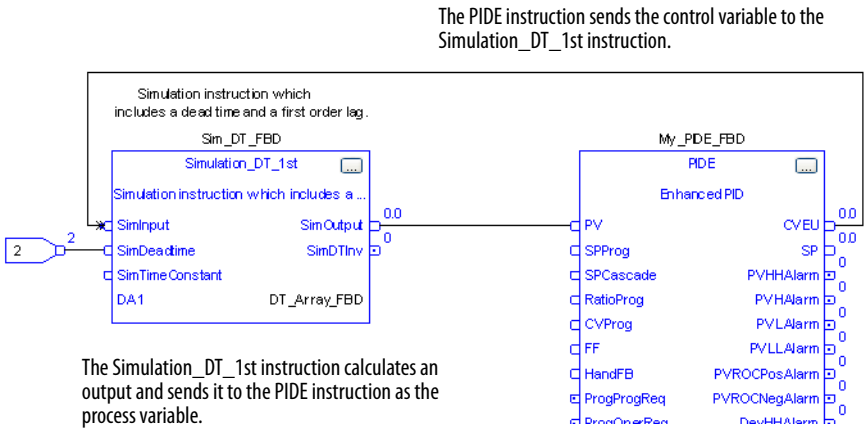
In this example, the instruction simulates a deadtime and lag (first order) process.

The Simulation_DT_1st instruction reads the control variable from the PID instruction. The PID instruction reads the SimOutput Parameter of the Simulation_DT_1st instruction.

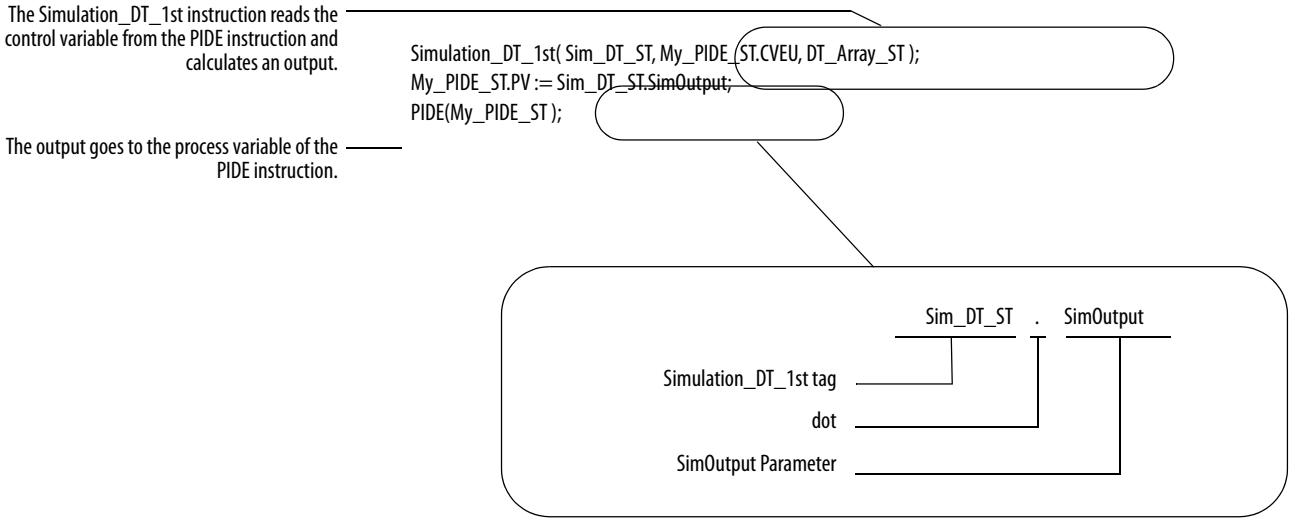


## Function Block Diagram Configuration

The PIDE instruction sends the control variable to the Simulation_DT_1st instruction. The Simulation_DT_1st instruction calculates an output and sends it to the PIDE instruction as the process variable




## Structured Text Configuration


The Simulation_DT_1st instruction reads the control variable from the PIDE instruction and calculates an output.
Simulation_DT_1st( Sim_DT_ST, My_PIDE_ST.CVEU, DT_Array_ST );
My_PIDE_ST.PV := Sim_DT_ST.SimOutput;
PIDE(My_PIDE_ST );
The output goes to the process variable of the PIDE instruction.
Sim_DT_ST
.
SimOutput
Simulation_DT_1st tag
dot
SimOutput Parameter

## Notes:

# Using Add-On Instructions

## Introduction

Add-On Instructions are used in your routines like any built-in instructions. You add calls to your instruction and then define the arguments for any parameters.

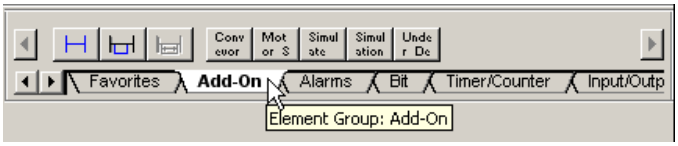

| Topic | Page |
|---|---|
| Accessing Add-On Instructions | 71 |
| Using the Add Element Dialog Box | 72 |
| Including an Add-On Instruction in a Routine | 73 |
| Tips for Using an Add-On Instruction | 75 |
| Programmatically Access a Parameter | 75 |
| Monitoring the Value of a Parameter | 77 |
| Viewing Logic and Monitor with Data Context | 79 |
| Determining if the Add-On Instruction is Source Protected | 81 |
| Copying an Add-On Instruction | 82 |
| Storing Your Add-On Instructions | 82 |


## Accessing Add-On Instructions

The Add-On Instruction can be used in any one of the Ladder Diagram, Function Block, or Structured Text languages (including Structured Text within Sequential Function Chart actions). The appearance of the instruction conforms to the language in which it is placed.

The Add-On Instructions in the project can be accessed from any of the normal instruction selection tools.

The instruction toolbar has an Add-On tab that lists all of the currently available Add-On Instructions in the project.

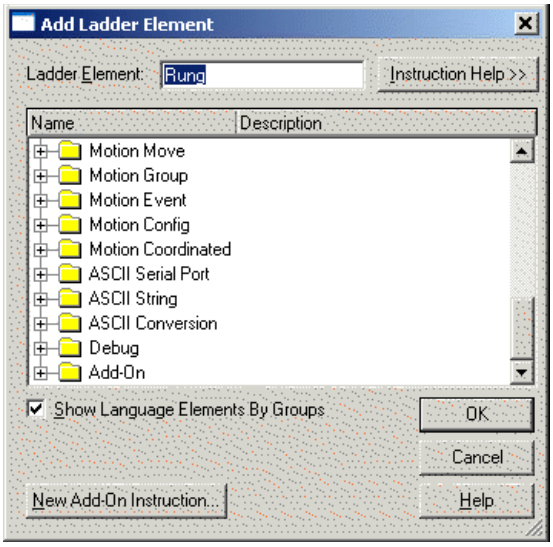



| IMPORTANT | Safety Add-On Instructions can be used only in safety routines, which are currently restricted to ladder logic. Safety Add-On Instructions are shown in the Language Element Toolbar only when the routine is a safety routine. |
|---|---|

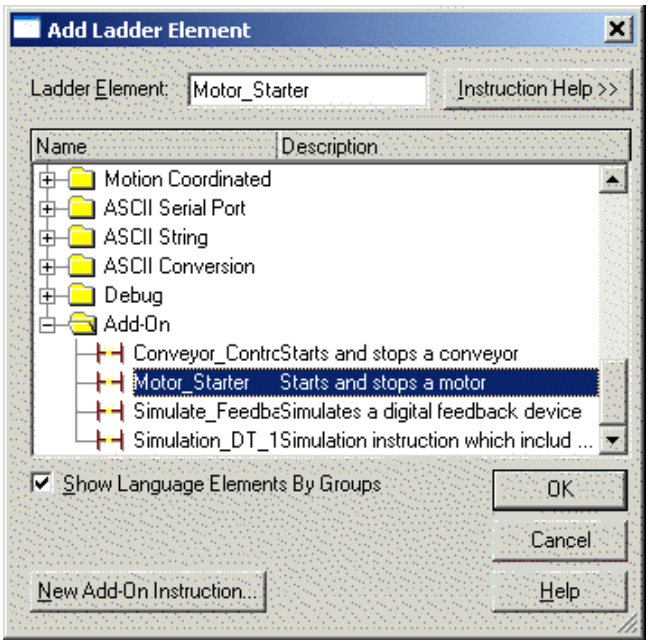
## Using the Add Element Dialog Box

Follow these instructions to access the Add (language) Element dialog box.

1. Press **Alt + Insert** anywhere in the editor or by right-clicking the logic in the Editor.

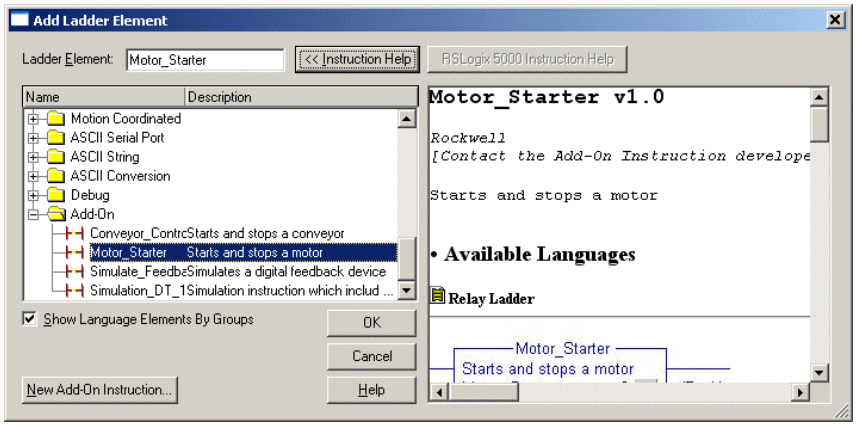



2. Locate the Add-On Instruction you want to add to your routine.



3. Select the Add-On Instruction and click **OK**.

   Use [Instruction Help >>] to display the instruction help for any instruction in the browser.




## Including an Add-On Instruction in a Routine

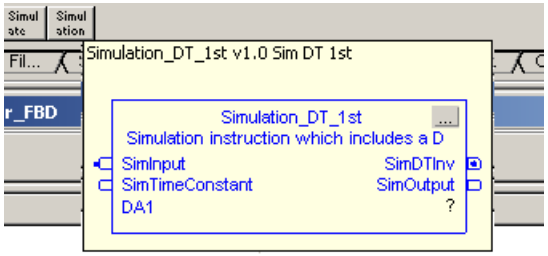
Follow this procedure when you want to use an Add-On Instruction in one of your routines.

1. Open the **Add-On Instruction** folder in the Controller Organizer and view the listed instructions.

   If the instruction you want to use is not listed, you need to do one of the following:

   - Create the instruction in your project.
   - Copy and paste an instruction into your project.
   - Get the file for an exported instruction definition and then import the instruction into your current project.

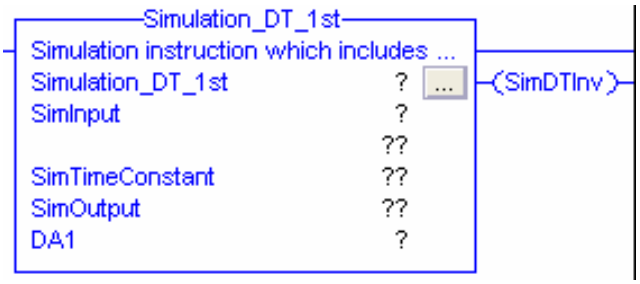
2. Open the routine that will use the instruction.
3. Click the **Add-On** tab on the instruction toolbar.
4. Click the desired Add-On Instruction, for example Simulation, and drag the instruction from the toolbar into the routine.




5. Define arguments for each Parameter on the instruction call.

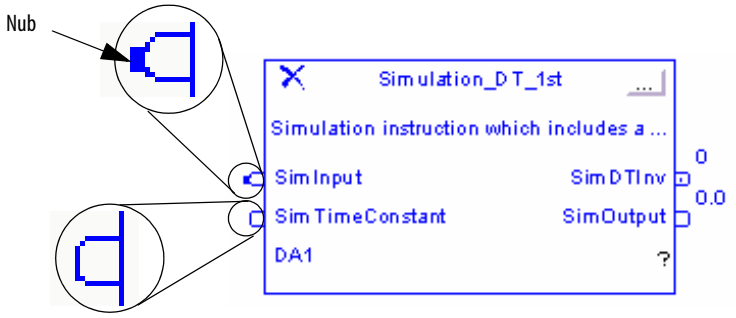
The instruction appears as follows in each of the languages.

**Ladder Diagram**



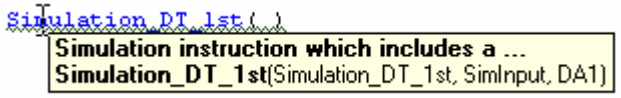

| Parameter With | Description |
|---|---|
| Single question mark | This is a required InOut parameter. Enter a tag. |
| Single and double question marks | This is a required Input or Output parameter. Enter a tag. |
| Double question marks | This is not a required parameter. You can either:<br>• Leave as is and use the default value.<br>• Enter a different value if it is an Input parameter. |

**Function Block Diagram**




| Item | Description |
|---|---|
| Nub on the end of pin | This is a required Input or Output parameter.<br>You must wire the pin to an IREF, OREF, connector, or another block to verify. |
| Single question mark | This is a required InOut parameter. Enter a tag. |
| No nub on the end of pin | This is not a required parameter.<br>You can either:<br>• Leave as is and use the default value.<br>• Enter a different value if it's an Input parameter. |

**Structured Text**



The instruction expects arguments for required parameters as listed in the instruction tooltip.

TIP For help with an instruction, select the instruction and then press **F1**. In Structured Text, make sure the cursor is in the blue instruction name.

## Tips for Using an Add-On Instruction

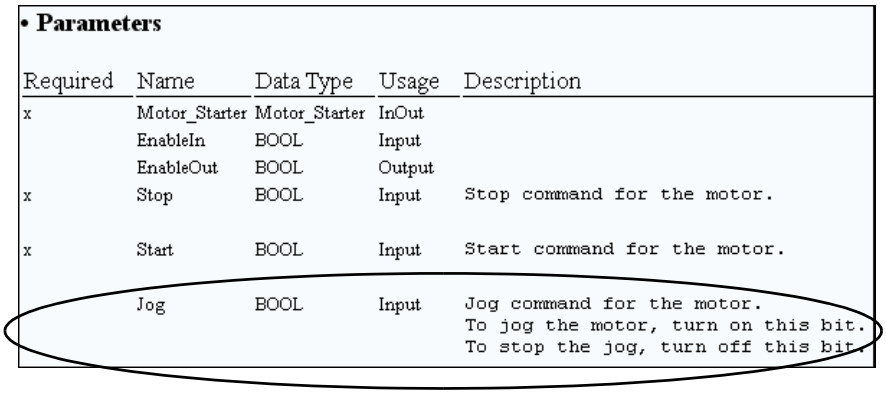
This table describes programming tips for you to reference when using Add-On Instructions.

| Topic | Description |
|---|---|
| Instruction Help | Use the instruction help to determine how to use the instruction in your code. |
| Ladder Rungs | In a ladder rung, consider if the instruction should be executed on a false rung condition. It may improve scan time to not execute it. |
| Data Types | A data type defined with the Add-On Instruction is used for the tag that provides context for the execution from your code. A tag must be defined of this Add-On Instruction-defined type on the call to the instruction. |
| Indexed Tag | You can use an indirect array indexed tag for the Instruction instance. One drawback is that you cannot monitor the Add-On Instruction by using this as a data context. |
| Passing Data | •Input and Output parameters are passed by value.<br>•InOut parameters are passed by reference. |

## Programmatically Access a Parameter

Follow these procedures for any language when you want to access an Add-On Instruction parameter that is not available on the instruction face by default.

The following procedures demonstrate how to use the Jog parameter of the Motor Starter Add-On Instructions.

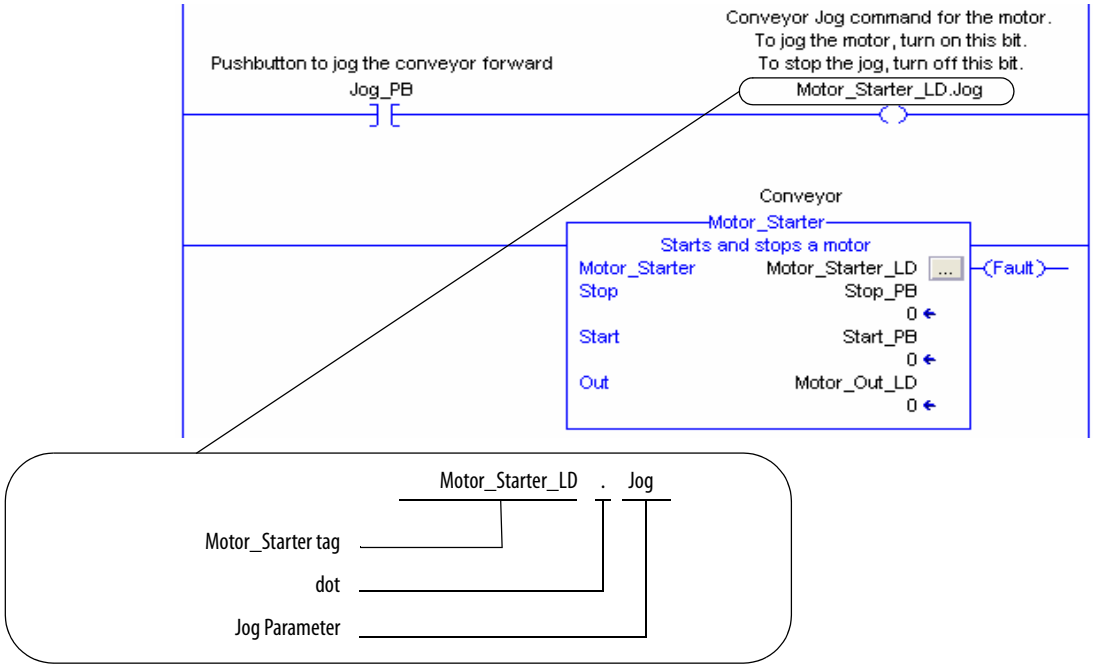


**• Parameters**

| Required | Name | Data Type | Usage | Description |
|---|---|---|---|---|
| x | Motor_Starter | Motor_Starter | InOut | |
| | EnableIn | BOOL | Input | |
| | EnableOut | BOOL | Output | |
| x | Stop | BOOL | Input | Stop command for the motor. |
| x | Start | BOOL | Input | Start command for the motor. |
| | Jog | BOOL | Input | Jog command for the motor.<br>To jog the motor, turn on this bit.<br>To stop the jog, turn off this bit. |

## Using the Jog Command in Ladder Diagram

The first rung sets the Jog bit of Motor_Starter_LD = Jog_PB.



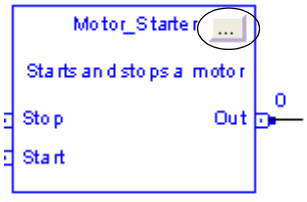

| |
|---|
| Use another instruction, an assignment, or an expression to read or write to the tag name of the parameter. Use this format for the tag name of the parameter.<br>*Add_On_Tag.Parameter* |

| Where | Is |
|---|---|
| *Add_On_Tag* | An instance tag defined by the Add On data type. |
| *Parameter* | Name of the parameter. |

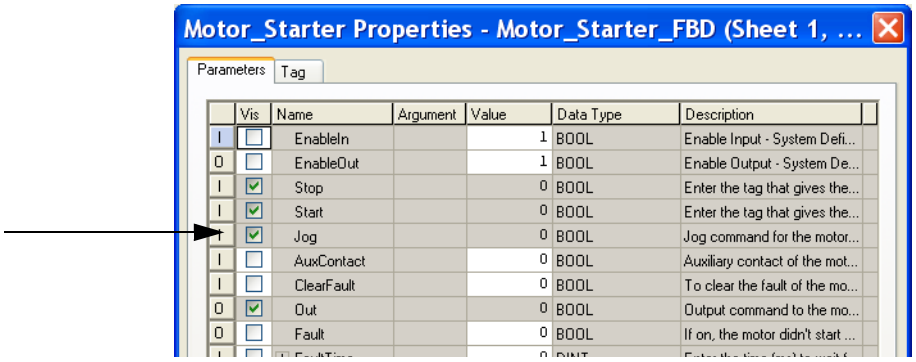
## Using the Jog Command In Function Block Diagram

Any parameter can be made visible or invisible except those defined as required. Required parameters are always visible. If the parameter is required, you will see it checked in the Properties dialog box.

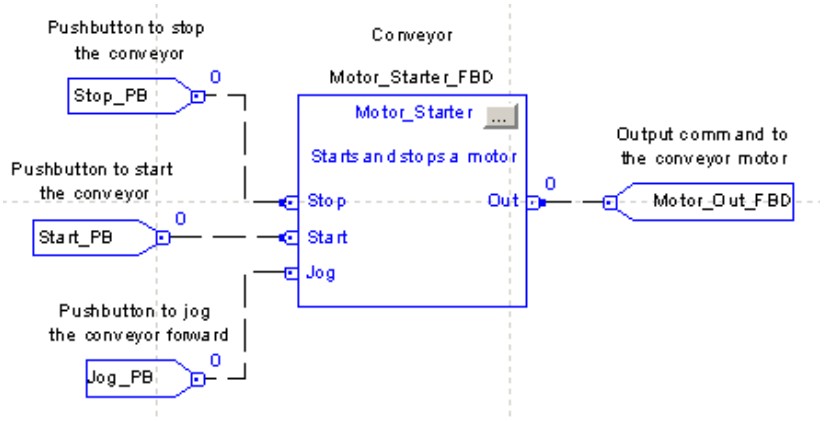
1. Click **Properties** for the instruction.



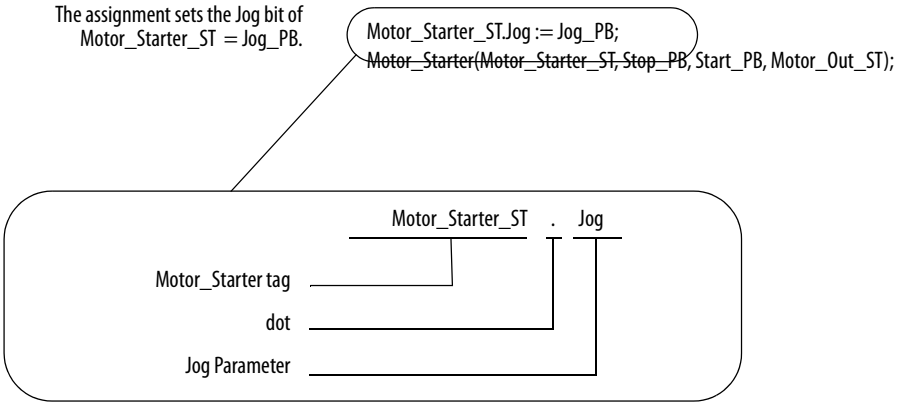

2. Check the **Vis** checkbox of the Jog parameter to use it in your diagram.
3. Click **OK**.




4. Wire to the pin for the parameter.




### Using the Jog Command in Structured Text

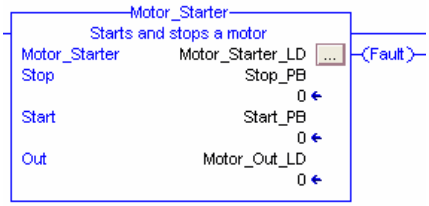



## Monitoring the Value of a Parameter

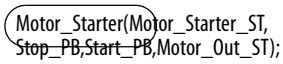
Follow this procedure when you want to see or change a parameter value of an Add-On Instruction.

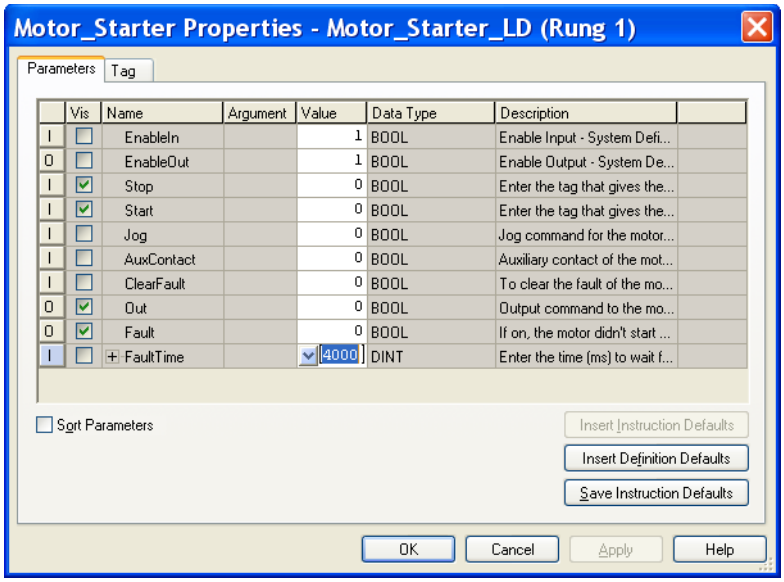
1. Open the Properties of the instruction based on what language you are using.

a. For either a Function Block or Ladder Diagram, click **Properties** ... for the instruction.




b. For Structured Text, right-click the instruction name and choose **Properties**.



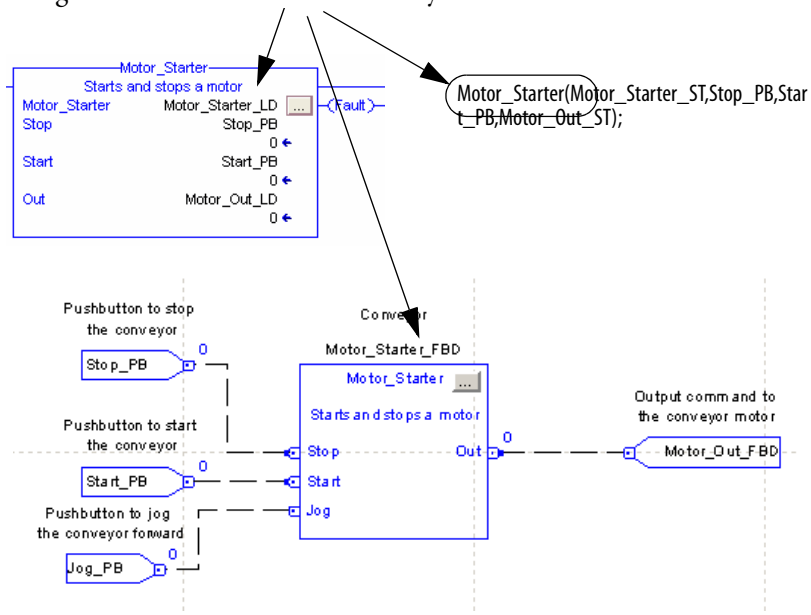


Motor_Starter Properties - Motor_Starter_LD (Rung 1)
Parameters
Tag
Vis
Name
Argument
Value
Data Type
Description
I EnableIn 1 BOOL Enable Input - System Defi...
O EnableOut 1 BOOL Enable Output - System De...
I Stop 0 BOOL Enter the tag that gives the...
I Start 0 BOOL Enter the tag that gives the...
I Jog 0 BOOL Jog command for the motor...
I AuxContact 0 BOOL Auxiliary contact of the mot...
I ClearFault 0 BOOL To clear the fault of the mo...
O Out 0 BOOL Output command to the mo...
O Fault 0 BOOL If on, the motor didn't start ...
I FaultTime 4000 DINT Enter the time (ms) to wait f...
Sort Parameters
Insert Instruction Defaults
Insert Definition Defaults
Save Instruction Defaults
OK
Cancel
Apply
Help

2. Monitor the value of the parameters and change any if needed.
3. Type a new value for each parameter as needed.
4. Click **Apply** and when finished, **OK**.

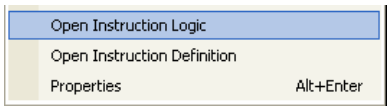
## Viewing Logic and Monitor with Data Context

Follow this procedure when you want to view the logic of an Add-On Instruction and monitor data values with the logic.

1. Right-click the instruction call in any routine.



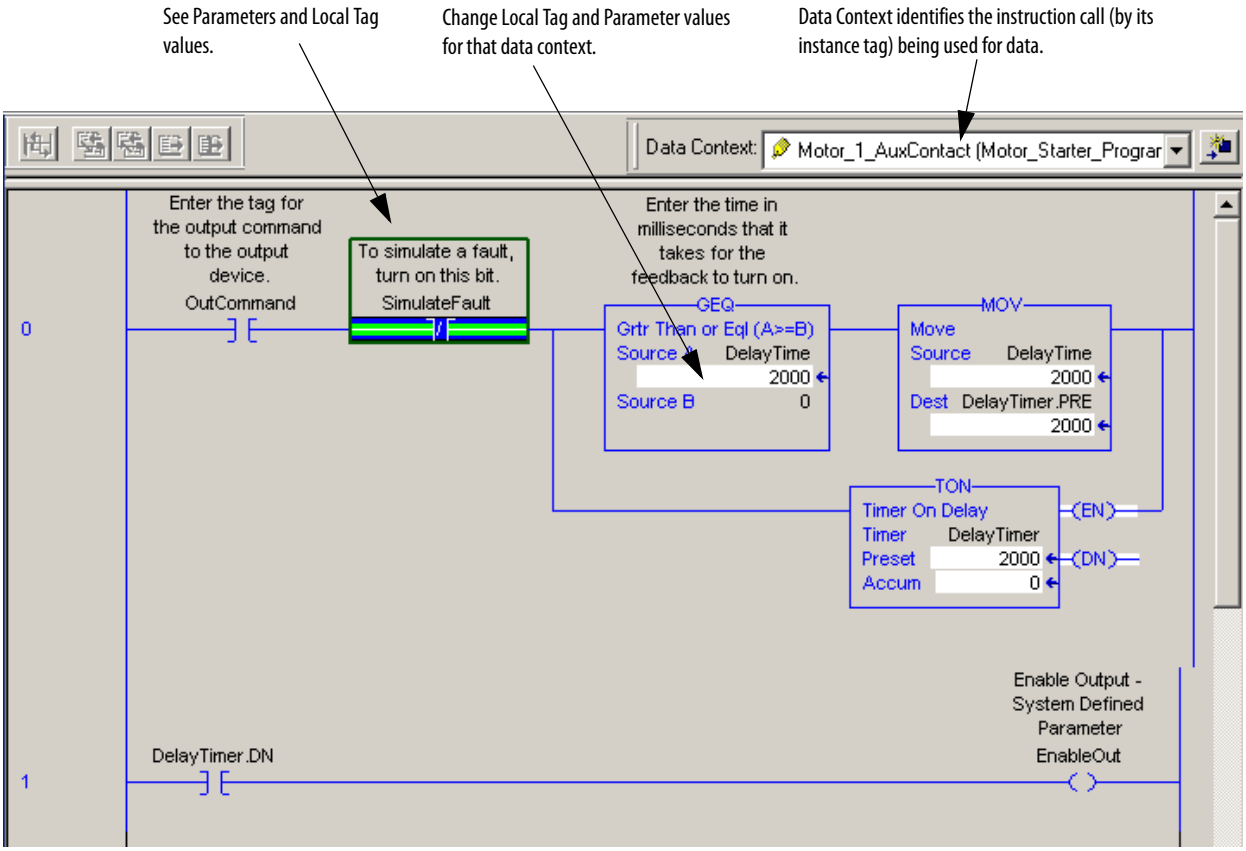

2. Choose **Open Instruction Logic**.



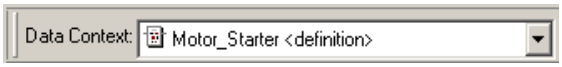

   The Language Editor opens with the Add-On Instruction's logic routine and with data values from the instruction call.

As you view the logic you can:

- Identify the instruction call whose tags are being used for data.
- See the logic as it executes (when online).
- See Parameter and Local Tag values.
- Change local tag and parameter values for the data instance selected.


See Parameters and Local Tag values.
Change Local Tag and Parameter values for that data context.
Data Context identifies the instruction call (by its instance tag) being used for data.
Data Context:
Motor_1_AuxContact (Motor_Starter_Prograr
Enter the tag for the output command to the output device.
OutCommand
To simulate a fault, turn on this bit.
SimulateFault
Enter the time in milliseconds that it takes for the feedback to turn on.
0
GEQ
Grtr Than or Eql (A>=B)
Source DelayTime
2000
Source B 0
MOV
Move
Source DelayTime
2000
Dest DelayTimer.PRE
2000
TON
Timer On Delay
Timer DelayTimer
Preset 2000
Accum 0
(EN)
(DN)
Enable Output - System Defined Parameter
EnableOut
DelayTimer.DN
1

To edit the logic of the Add-On Instruction, you must select the instruction <definition> in Data Context.




You cannot edit the instruction logic:

- Online.
- When the logic is in the context of an instruction call.
- If the instruction is source-protected.
- If the instruction is sealed with an instruction signature.

## Determining if the Add-On Instruction is Source Protected

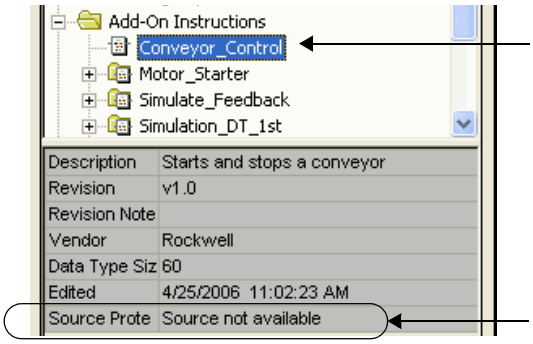
An Add-On Instruction may be source protected so you cannot view the logic. Follow these steps to see if an Add-On Instruction is source protected.

1. Select the Add-On Instruction in the Controller Organizer.

   The Add-On Instruction cannot be expanded when fully protected.

2. Look in the Quick View pane for Source Protection.

   If the Source Protection attribute is not listed, then the instruction is not protected.



## Copying an Add-On Instruction

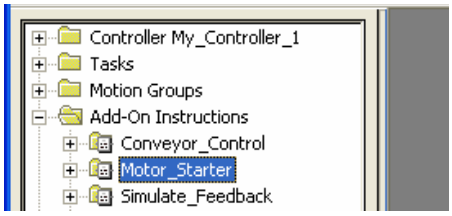
You can copy an Add-On Instruction into your project when it exists in another Logix Designer project. After you copy the Add-On Instruction, you can use the instruction as is or rename it, modify it, and then use it in your programs.

| | |
|---|---|
| IMPORTANT | Use caution when copying and pasting components between different versions of the Logix Designer application. Logix Designer software only supports pasting to the same version or newer version of Logix Designer application. Pasting to an earlier version of RSLogix 5000 software is not supported. When pasting to an earlier version, the paste action may succeed, but the results may not be as expected. |

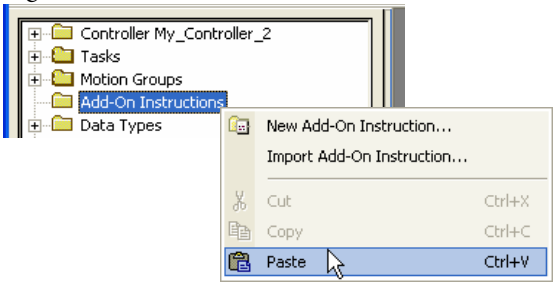
| | |
|---|---|
| TIP | When copying and pasting Add-On Instructions, consider these guidelines:<br>• You cannot paste a safety Add-On Instruction into a standard routine.<br>• You cannot paste a safety Add-On Instruction into a safety project that has been safety-locked or one that has a safety task signature.<br>• You cannot copy and paste a safety Add-On Instruction while online. |

Follow these steps to copy the Add-On Instruction.

1. Open the Logix Designer project that contains the Add-On Instruction.
2. Find the instruction in the Add-On Instructions folder.




3. Right-click the instruction and choose **Copy**.
4. Go to the other project where you want to paste the instruction.
5. Right-click the Add-On Instructions folder and choose **Paste**.




## Storing Your Add-On Instructions

There are two ways to store a group of Add-On Instructions together. One is to save your Add-On Instructions in a project file. Another is to create an L5X export file, as described in Chapter 4.

Follow these steps to store your instructions by saving them in a project file.

1. Identify what instructions you want to store.
2. Place them in a project file called something like *MyInstructions.ACD*.

3. Open other projects in additional instances of the application and use copy and paste or drag and drop to move a copy of the instruction from *MyInstructions.ACD* to another project.

If any of these instructions reference the same Add-On Instruction or User-Defined Data Type, there is only one shared copy in the project file. When an Add-On Instruction is copied to another project, it also copies any instruction it references to the target project.

## Notes:

# Importing and Exporting Add-On Instructions

## Introduction


| Topic | Page |
|---|---|
| Creating an Export File | 85 |
| Importing an Add-On Instruction | 88 |
| Updating an Add-On Instruction to a Newer Revision via Import | 90 |


## Creating an Export File

When you choose to export an Add-On Instruction, the exported Add-On Instruction includes all of its parameters, local tags, and routines. These will be imported with the Add-On Instruction automatically.

Optionally, you can include any nested Add-On Instructions or User-Defined Data Types that are referenced by the exported Add-On Instruction. Referenced Add-On Instructions and data types are exported to the .L5X file if you select **Include all referenced Add-On Instructions and User-Defined Types** during the export.

Add-On Instruction definition references may also be exported when a program, routine, set of rungs, or User-Defined Data Type is exported.

TIP If an Add-On Instruction uses Message (MSG) instruction and InOut parameters of type MESSAGE, you may wish to export a rung containing the Add-On Instruction to include the MESSAGE tags. This captures the message configuration data, such as type and path.

In deciding how to manage your Add-On Instruction definitions in export files, you need to consider your goals in storing the definitionss .

| If | Then |
|---|---|
| You want to store many Add-On Instructions that share a set of common Add-On Instructions or User-Defined Data Types in a common location | Export to separate files as described on page 86. |
| You want to distribute an Add-On Instruction as one file | Export to a single file as described on page 87. |
| You want to manage each Add-On Instruction as a standalone instruction | |
| You want to preserve the instruction signature on your Add-On Instruction | |

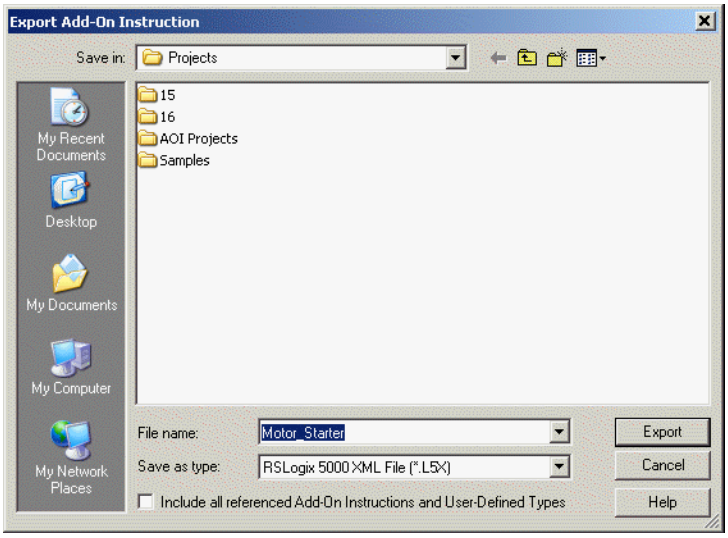
TIP Add-On Instructions with instruction signatures are encrypted upon export to prevent modifications to the export file.

## Exporting to Separate Files

If you want to store many Add-On Instructions that share a set of common Add-On Instructions or User-Defined Data Types in a common location, you may want to choose to export each Add-On Instruction and User-Defined Data Types to separate files without including references.

Follow these steps to export to separate files.

1. Right-click the Add-On Instruction in the Controller Organizer, and choose **Export Add-On Instruction**.
2. Select the common location to store the .L5X file.




3. Type a name for the file.
4. Clear **Include referenced Add-On Instructions and User-Defined Types**.
5. Click **Export**.
6. Follow the above steps to individually export the other shared Add-On Instructions and User-Defined Data Types.

Using export in this way lets you manage the shared Add-On Instruction and User-Defined Data Types independently of the Add-On Instructions that reference them. One advantage of this is the ability to update the shared component without having to regenerate all the export files for the instructions that reference it. That is, it is only stored in one file instead of in every file whose instruction references it. This can help with the maintenance of the instructions as you only have to update one export file.

To use Add-On Instructions that have been exported in a separate file, without references, you must first import any User-Defined Data Types of Add-On Instructions that the exported instruction references before the import of the referencing instruction can be successful. To do this, work from the bottom up.

Import the lowest-level User-Defined Data Types and any User-Defined Data Types that reference them.

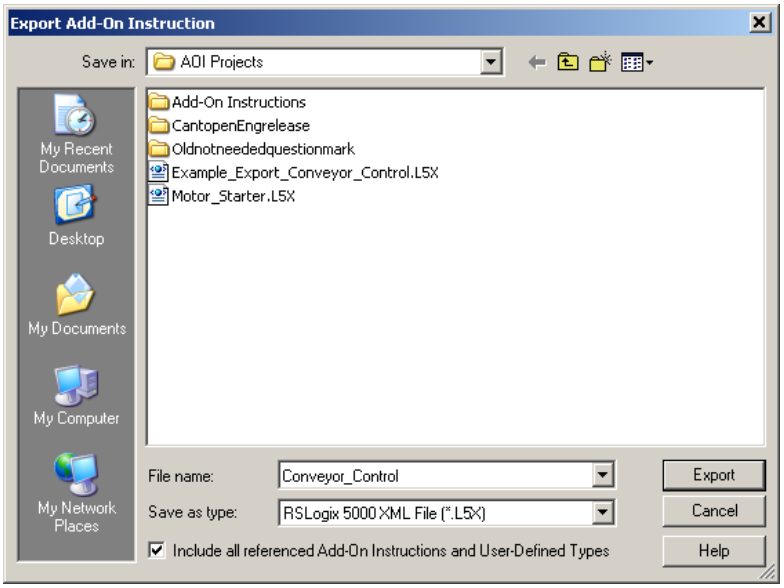
Then, import the lowest-level Add-On Instructions, followed by any Add-On Instructions that reference those low-level Add-On Instructions. Once all of the items it references are in place, the import of the Add-On Instruction will succeed.

## Exporting to a Single File

If you manage each Add-On Instruction as a standalone, you may want to export the instruction and any referenced Add-On Instructions or User-Defined Data Types into one export file. By including any referenced Add-On Instructions or User-Defined Data Types, you also make it easier to preserve the instruction signature of an Add-On Instruction.

Follow these steps to export to a single file and include any referenced items.

1. Right-click **Add-On Instruction** in Controller Organizer and choose Export Add-On Instruction.
2. Choose the location to store the .L5X file.




3. Type a name for the file.
4. Check **Include referenced Add-On Instructions and User-Defined Types**.
5. Click **Export**.

This exports the selected Add-On Instruction and all the referenced instructions into the same export file. This file can be used to distribute an Add-On Instruction. When the exported Add-On Instruction is imported into the project, the referenced instructions are imported as well in one step.

## Importing an Add-On Instruction

You can import an Add-On Instruction that was exported from another RSLogix 5000 project. When importing an Add-On Instruction, the parameters, local tags, and routines are imported as part of the Add-On Instruction. Once the project has the Add-On Instruction, you can use it in your programs.

### Import Considerations

**ATTENTION: Editing an .L5K or .L5X File**

**ATTENTION:** The EditedDate attribute of an Add-On Instruction must be updated if the Add-On Instruction is modified by editing an .L5K or .L5X file. If the Logix Designer application detects edits to the Add-On Instruction, but the Edited Date attribute is the same, the Add-On Instruction will not be imported.

When importing Add-On Instructions directly or as references, consider these guidelines.

| Topic | Consideration |
|---|---|
| Tag Data | Imported tags that reference an Add-On Instruction in the import file may be affected if the Add-On Instruction is not imported as well. In this case, the imported tag's data may be converted if the existing Add-On Instruction's data structure is different and tag data may be lost.<br>If an existing Add-On Instruction is overwritten, project tag data may be converted if the Add-On Instruction's data structure is different and tag data may be lost.<br>See "Import Configuration" on page 89 for more information. |
| Logic | Imported logic that references the Add-On Instruction in the import file may be affected if the Add-On Instruction is not imported. If an existing Add-On Instruction is used for the imported logic reference and the parameter list of the Add-On Instruction in the project is different, the project may not verify or it may verify but not work as expected.<br>If an existing Add-On Instruction is overwritten, logic in the project that references the Add-On Instruction may be affected. The project may not verify or may verify but not work as expected.<br>See "Import Configuration" on page 89 for more information. |
| Add-On Instructions While Online | An Add-On Instruction cannot be overwritten during import while online with the controller, although a new Add-On Instruction may be created while online. |
| Final Name Change | If the Final Name of an Add-On Instruction is modified during import configuration, the edit date of the imported Add-On Instruction will be updated. In addition, all logic, tags, User-Defined Data Types, and other Add-On Instructions in the import file that reference the Add-On Instruction will be updated to reference the new name. As a result, the edit date of any Add-On Instruction that references the Add-On Instruction will be updated.<br>Add-On Instructions that have been sealed with an instruction signature cannot be renamed during import. |
| User-Defined Data Types | Add-On Instructions cannot overwrite User-Defined Data Types. Add-On Instructions and User-Defined Data Types must have unique names. |
| Instruction Signature | If you import an Add-On Instruction with an instruction signature into a project where referenced Add-On Instructions or User-Defined Data Types are not available, you may need to remove the signature.<br>You can overwrite an Add-On Instruction that has an instruction signature by importing a different Add-On Instruction with the same name into an existing routine. Add-On Instructions that have been sealed with an instruction signature cannot be renamed during import. |
| Safety Add-On Instructions | You cannot import a safety Add-On Instruction into a standard task.<br>You cannot import a safety Add-On Instruction into a safety project that has been safety-locked or one that has a safety task signature.<br>You cannot import a safety Add-On Instruction while online.<br>Class, instruction signature, signature history, and safety instruction signature, if it exists, remain intact when an Add-On Instruction with an instruction signature is imported. |

| IMPORTANT | Importing an Add-On Instruction created in version 18 or later of RSLogix 5000 software into an older project that does not support Add-On Instruction signatures causes the Add-On Instruction to lose attribute data and the instruction may no longer verify. |
|---|---|

## Import Configuration

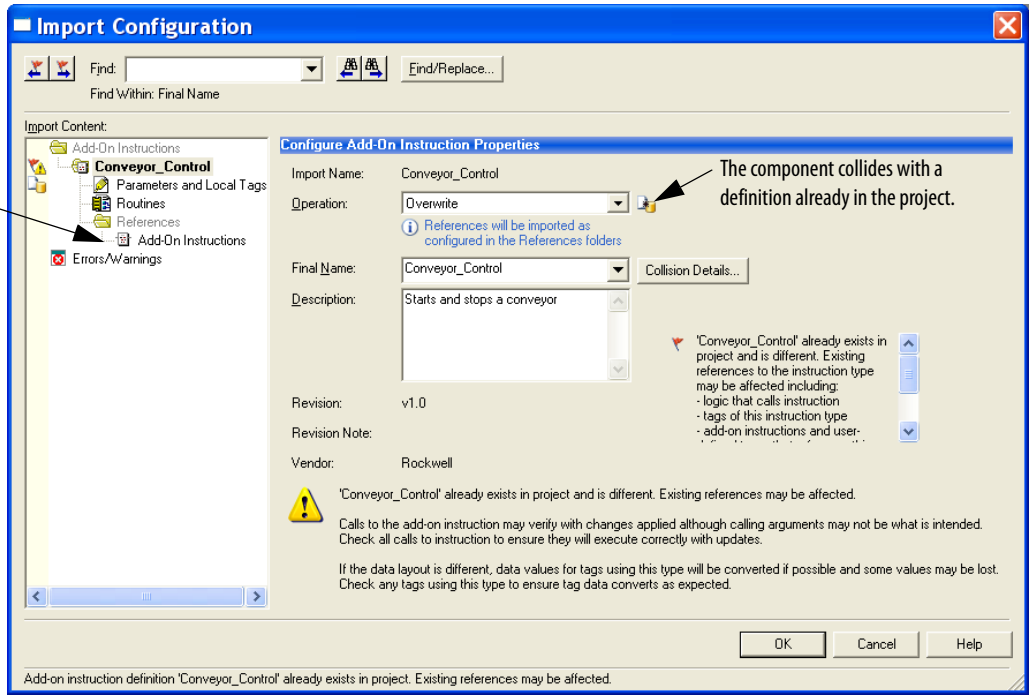
When you select a file to import, the Import Configuration dialog box lets you choose how the Add-On Instruction and referenced components are imported.

If there are no issues, you can simply click **OK** to complete the import.

If your Add-On Instruction collides with one already in the project, you can:

- Rename it, by typing a new, unique name in the Final Name field.
- Choose **Overwrite** from the Operation menu.
- Choose **Use Existing** from the Operation menu.

Click **Add-On Instructions** to see the Configure Add-On Instruction References view. If any referenced instruction definitions collide, the References folder is flagged.


Import Configuration
Find:
Find/Replace...
Find Within: Final Name
Import Content:
Add-On Instructions
Conveyor_Control
Parameters and Local Tags
Routines
References
Add-On Instructions
Errors/Warnings
Configure Add-On Instruction Properties
Import Name: Conveyor_Control
Operation: Overwrite
References will be imported as configured in the References folders
The component collides with a definition already in the project.
Final Name: Conveyor_Control
Collision Details...
Description: Starts and stops a conveyor
'Conveyor_Control' already exists in project and is different. Existing references to the instruction type may be affected including:
- logic that calls instruction
- tags of this instruction type
- add-on instructions and user-
Revision: v1.0
Revision Note:
Vendor: Rockwell
'Conveyor_Control' already exists in project and is different. Existing references may be affected.
Calls to the add-on instruction may verify with changes applied although calling arguments may not be what is intended. Check all calls to instruction to ensure they will execute correctly with updates.
If the data layout is different, data values for tags using this type will be converted if possible and some values may be lost. Check any tags using this type to ensure tag data converts as expected.
OK
Cancel
Help
Add-on instruction definition 'Conveyor_Control' already exists in project. Existing references may be affected.

| TIP | You can only rename an Add-On Instruction if it has not been sealed with an instruction signature.<br><br>To rename an Add-On Instruction that has been source-protected, you need the source key. |
|---|---|

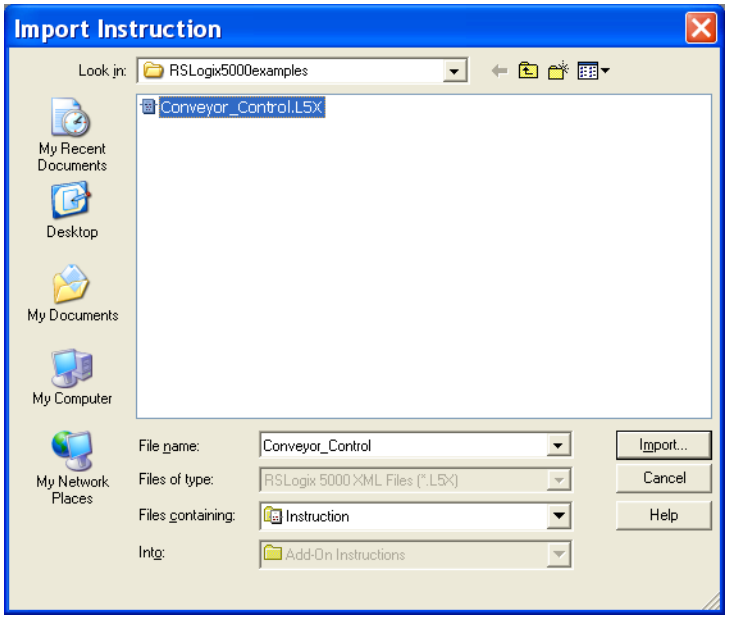
The Collision Details button lets you view the Property Compare tab, which shows the differences between the two instructions, and the Project References tab, which shows where the existing Add-On Instruction is used.

## Updating an Add-On Instruction to a Newer Revision via Import

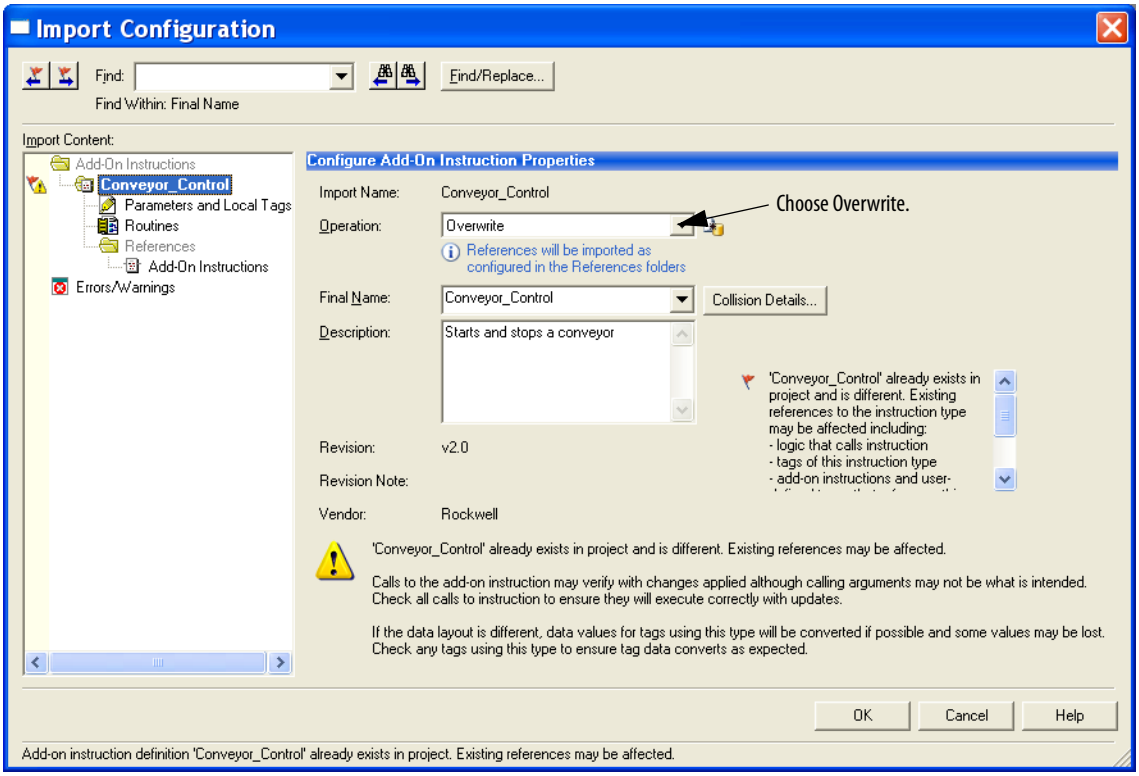
When you need to update an instruction to a newer revision, you can import it from an .L5X file or copy it from an existing project. You must be offline to update an Add-On Instruction.

Follow these steps to update an Add-On Instruction to a newer revision by importing it.

1. Right-click the **Add-On Instruction** folder and choose **Import Add-On Instruction**.
2. Select the file with the Add-On Instruction and click **Import**.

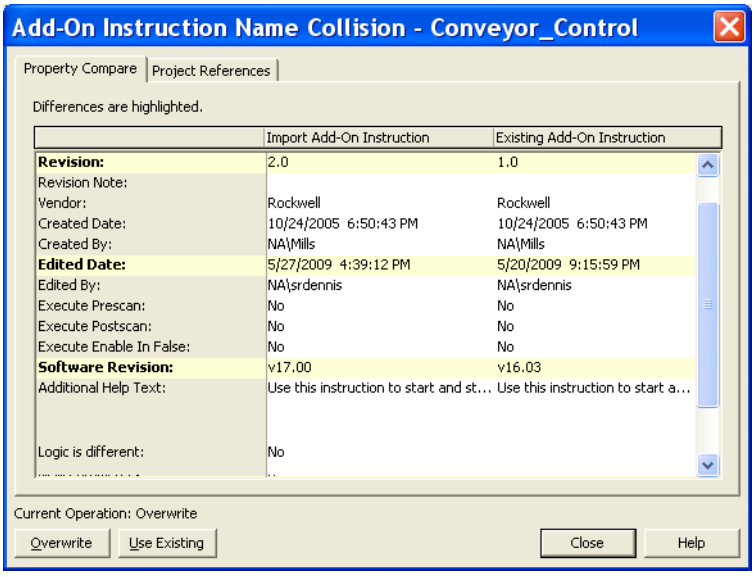


3. Review the Import Configuration dialog box, and from the Operations menu, choose **Overwrite**.


Import Configuration
Find:
Find/Replace...
Find Within: Final Name
Import Content:
Add-On Instructions
Conveyor_Control
Parameters and Local Tags
Routines
References
Add-On Instructions
Errors/Warnings
Configure Add-On Instruction Properties
Import Name:
Conveyor_Control
Operation:
Overwrite
Choose Overwrite.
References will be imported as configured in the References folders
Final Name:
Conveyor_Control
Collision Details...
Description:
Starts and stops a conveyor
'Conveyor_Control' already exists in project and is different. Existing references to the instruction type may be affected including:
- logic that calls instruction
- tags of this instruction type
- add-on instructions and user-
Revision:
v2.0
Revision Note:
Vendor:
Rockwell
'Conveyor_Control' already exists in project and is different. Existing references may be affected.
Calls to the add-on instruction may verify with changes applied although calling arguments may not be what is intended. Check all calls to instruction to ensure they will execute correctly with updates.
If the data layout is different, data values for tags using this type will be converted if possible and some values may be lost. Check any tags using this type to ensure tag data converts as expected.
OK
Cancel
Help
Add-on instruction definition 'Conveyor_Control' already exists in project. Existing references may be affected.

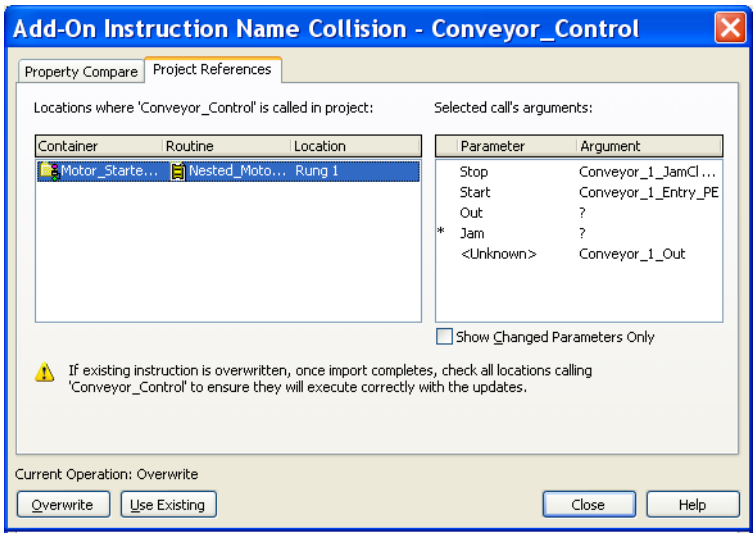
4. Click **Collision Details** to see any differences in the Add-On Instructions and to view where the Add-On Instruction is used.

   The Property Compare tab shows the differences between the instructions, in this case, the Revision, Edited Date, and Software Revision.


Add-On Instruction Name Collision - Conveyor_Control
Property Compare
Project References
Differences are highlighted.
Import Add-On Instruction
Existing Add-On Instruction
Revision: 2.0 1.0
Revision Note:
Vendor: Rockwell Rockwell
Created Date: 10/24/2005 6:50:43 PM 10/24/2005 6:50:43 PM
Created By: NA\Mills NA\Mills
Edited Date: 5/27/2009 4:39:12 PM 5/20/2009 9:15:59 PM
Edited By: NA\srdennis NA\srdennis
Execute Prescan: No No
Execute Postscan: No No
Execute Enable In False: No No
Software Revision: v17.00 v16.03
Additional Help Text: Use this instruction to start and st... Use this instruction to start a...
Logic is different: No
Current Operation: Overwrite
Overwrite
Use Existing
Close
Help

TIP The Compare dialog box only compares extended properties for each instruction definition, such as description, revision, or edited date. For effective revision control, enter a detailed revision note.

The Project References tab shows where the existing Add-On Instruction is used.




IMPORTANT Check each routine where your Add-On Instruction is used to make sure that your existing program code will work correctly with the new version of the instruction.

For more information on updates to arguments, see "Updates to Arguments Following Parameter Edits" on page 41.

5. Click **Close** and then **OK** to complete the operation.

## A

**access**
- **Add-On Instructions** 71

**Add Element dialog box** 72
**Add-On Instruction Definition Editor** 37
**alias**
- **parameter** 28, 37

**array** 28

## C

**Change History tab** 17
**changing class** 53
**class** 13
- **changing** 53

**collision**
- **import** 89, 91

**compare instructions** 91
**constant value tags** 31
**copy**
- **Add-On Instruction** 82
- **default values** 43
- **instruction signature** 56
- **safety Add-On Instruction** 82
- **safety instruction signature** 57
- **signature history** 57

**Copy Default Values dialog box** 44
**create**
- **Add-On Instruction** 35
- **alias parameter** 37
- **EnableInFalse routine** 50
- **instruction help** 61
- **instruction signature** 56
- **local tags** 39
- **logic** 44
- **parameters** 37
- **postscan routine** 49
- **prescan routine** 47
- **signature history** 57

## D

**data access** 24
**data access control** 30
**data context views** 54
**data types**
- **alias** 28
- **parameters** 27, 37
- **size** 22
- **tags** 40

**default values** 38, 40
- **copy** 43

**delete**
- **safety instruction signature** 58

## E

**Edit Tags tab** 41
**EnableIn parameter** 51
- **Function Block Diagram** 52
- **ladder diagram** 52
- **structured text** 52

**EnableInFalse routine** 50
- **create** 50

**EnableOut parameter** 51
**export** 85-88
**external access** 31

## F

**Function Block Diagram**
- **instruction example** 74

## G

**General tab** 13
**generate**
- **instruction signature** 56

**GSV** 26

## H

**help**
- **create** 61
- **example** 62

**Help tab** 18

## I

**import** 88-92
**Import Configuration dialog box** 91
**instruction**
- **size** 22
- **toolbar** 71

**instruction signature**
- **changing** 20
- **copy** 56
- **definition** 16
- **generate** 56
- **language switching** 20, 63
- **remove** 56
- **restricted actions** 16, 20
- **source protection** 20

## L

**Ladder Diagram**
- **instruction example** 74

**language switching** 63
- **instruction signature** 20

**Last Edit Date** 57
**local tags**
- **create** 39
- **external access** 40
- **planning** 33

**Local Tags tab** 14
- **create tags** 40

**logic** 44

**execution** 44

## M

**monitor**
**data values** 79
**parameter values** 77
**move**
**parameter** 38

## N

**naming conventions** 32, 36
**nesting** 22
**data access** 24
**planning** 33

## O

**object classes** 26

## P

**parameter**
**alignment** 41-42
**parameters**
**alias** 28
**create** 37
**EnableIn** 51
**EnableOut** 51
**monitor** 77
**move** 38
**planning** 32
**reorder** 38
**required** 29
**visible** 29, 30
**Parameters tab** 14
**create a parameter** 37
**passing arguments** 27
**performance** 45
**planning** 32
**postscan routine** 48
**create** 49
**prescan routine** 46
**create** 47
**programming language**
**choosing** 21
**planning** 33
**programming tips** 75
**Project References tab** 92
**Property Compare tab** 91

## R

**remove**
**instruction signature** 56
**reorder parameters** 38
**required parameters** 29
**routine**
**EnableInFalse** 50
**postscan** 48
**prescan** 46

## S

**safety**
**class** 13
**restrictions** 19
**tags** 30
**safety Add-On Instruction**
**copy** 82
**import** 88
**safety application instructions**
**restrictions** 25
**safety instruction signature** 16, 21
**copy** 57
**create** 58
**delete** 58
**invalid** 58
**view** 59
**safety task signature** 19, 53, 82, 88
**additional resources** 8
**Scan Mode tab** 16
**scan modes** 45-51
**planning** 34
**verify** 55
**Scan Modes tab** 46
**SFC Action** 48
**signature history** 17, 57
**Signature tab** 16
**SIL 3** 21
**source protection**
**applying** 56
**enabling** 56
**instruction signature** 20
**options** 56
**planning** 32
**Quick View pane** 81
**SSV** 26
**standard**
**class** 13
**tags** 30
**store your instructions** 82
**Structured Text**
**instruction example** 74

## T

**tags**
**create local tags** 39
**standard and safety** 30
**test** 53
**planning** 34
**transitional instructions** 21
**translation**
**See language switching.**

## U

**unavailable instructions** 24
**update Add-On Instruction revision** 90

## V

**visible**
**parameters** 29, 30

## Rockwell Automation Support

Rockwell Automation provides technical information on the Web to assist you in using its products.
At http://www.rockwellautomation.com/support/, you can find technical manuals, a knowledge base of FAQs, technical and application notes, sample code and links to software service packs, and a MySupport feature that you can customize to make the best use of these tools.

For an additional level of technical phone support for installation, configuration, and troubleshooting, we offer TechConnect support programs. For more information, contact your local distributor or Rockwell Automation representative, or visit http://www.rockwellautomation.com/support/.

### Installation Assistance

If you experience a problem within the first 24 hours of installation, review the information that is contained in this manual. You can contact Customer Support for initial help in getting your product up and running.

| | |
|---|---|
| United States or Canada | 1.440.646.3434 |
| Outside United States or Canada | Use the Worldwide Locator at http://www.rockwellautomation.com/support/americas/phone_en.html, or contact your local Rockwell Automation representative. |

### New Product Satisfaction Return

Rockwell Automation tests all of its products to ensure that they are fully operational when shipped from the manufacturing facility. However, if your product is not functioning and needs to be returned, follow these procedures.

| | |
|---|---|
| United States | Contact your distributor. You must provide a Customer Support case number (call the phone number above to obtain one) to your distributor to complete the return process. |
| Outside United States | Please contact your local Rockwell Automation representative for the return procedure. |

## Documentation Feedback

Your comments will help us serve your documentation needs better. If you have any suggestions on how to improve this document, complete this form, publication RA-DU002, available at http://www.rockwellautomation.com/literature/.

Rockwell Otomasyon Ticaret A.Ş., Kar Plaza İş Merkezi E Blok Kat:6 34752 İçerenköy, İstanbul, Tel: +90 (216) 5698400

**www.rockwellautomation.com**

**Power, Control and Information Solutions Headquarters**

Americas: Rockwell Automation, 1201 South Second Street, Milwaukee, WI 53204-2496 USA, Tel: (1) 414.382.2000, Fax: (1) 414.382.4444
Europe/Middle East/Africa: Rockwell Automation NV, Pegasus Park, De Kleetlaan 12a, 1831 Diegem, Belgium, Tel: (32) 2 663 0600, Fax: (32) 2 663 0640
Asia Pacific: Rockwell Automation, Level 14, Core F, Cyberport 3, 100 Cyberport Road, Hong Kong, Tel: (852) 2887 4788, Fax: (852) 2508 1846

Publication 1756-PM010E-EN-P - September 2012
Supersedes Publication1756-PM010D-EN-P - November 2011

Copyright © 2012 Rockwell Automation, Inc. All rights reserved. Printed in the U.S.A.